INTRODUCTION TO FORTRAN

NAVAL EDUCATION AND TRAINING COMMAND

RATE TRAINING MANUAL

NAVEDTRA 10078

Although the words "he", "him", and "his", are used sparingly in this manual to enhance communication, they are not intended to be gender driven nor to affront or discriminate against anyone reading *Introduction to FORTRAN*, NAVEDTRA 10078.

Naval Education And Training Program Development Center

INTRODUCTION TO FORTRAN

NEWARK PUBLIC LIBRARY
NOV 20 1980
U. S. GOVT. PUBS.
DEPOSITORY 376

NAVEDTRA 10078

DEPARTMENT OF THE NAVY
CHIEF OF NAVAL EDUC AND TNG

1980 Edition Prepared by
DPC Kenneth L. McDaniel

PREFACE

This Rate Training Manual is intended to serve as an aid for personnel who require the theoretical knowledge and skills of FORTRAN.

Designed for individual study or formal classroom instruction, the RTM provides subject matter that relates directly to the occupational qualifications. The OCC-ECC designed to accompany this RTM provides the usual way of satisfying the requirements for completing the RTM.

Programming experience is not required to complete this course. It is recommended that personnel applying for this course should have completed the *Data Processing Technician 3 & 2* course.

This training manual was prepared by the Naval Education and Training Program Development Center, Pensacola, Florida, for the Chief of Naval Education and Training.

1980 Edition

Stock Ordering No.
0502-LP-050-3900

Published by
NAVAL EDUCATION AND TRAINING PROGRAM
DEVELOPMENT CENTER

UNITED STATES
GOVERNMENT PRINTING OFFICE
WASHINGTON, D.C.: 1980

THE UNITED STATES NAVY

GUARDIAN OF OUR COUNTRY

The United States Navy is responsible for maintaining control of the sea and is a ready force on watch at home and overseas, capable of strong action to preserve the peace or of instant offensive action to win in war.

It is upon the maintenance of this control that our country's glorious future depends; the United States Navy exists to make it so.

WE SERVE WITH HONOR

Tradition, valor, and victory are the Navy's heritage from the past. To these may be added dedication, discipline, and vigilance as the watchwords of the present and the future.

At home or on distant stations we serve with pride, confident in the respect of our country, our shipmates, and our families.

Our responsibilities sober us; our adversities strengthen us.

Service to God and Country is our special privilege. We serve with honor.

THE FUTURE OF THE NAVY

The Navy will always employ new weapons, new techniques, and greater power to protect and defend the United States on the sea, under the sea, and in the air.

Now and in the future, control of the sea gives the United States her greatest advantage for the maintenance of peace and for victory in war.

Mobility, surprise, dispersal, and offensive power are the keynotes of the new Navy. The roots of the Navy lie in a strong belief in the future, in continued dedication to our tasks, and in reflection on our heritage from the past.

Never have our opportunities and our responsibilities been greater.

CONTENTS

CHAPTER Page

1. FORTRAN 1-1
2. Constants and Variables 2-1
3. Arithmetic 3-1
4. Input/Output Operations 4-1
5. Control Statements 5-1
6. Subscripted Variables and Arrays 6-1
7. Alphanumeric Data 7-1

APPENDIX

I. Glossary AI-1
II. Complete FORTRAN Problem AII-1

INDEX I-1

CHAPTER 1

FORTRAN

1.0 INTRODUCTION

This manual is intended to serve as an introduction to the FORTRAN (FORmula TRANslation) programming language. There is no prerequisite programming experience necessary for successful completion of this course, although exposure to an automatic data processing (ADP) environment may enhance the reader's understanding of the language.

The primary authoritative source for this manual is the American National Standard programming language FORTRAN publication (ANSI X3.9-1978). Direction for the use of this publication is provided in SECNAVINST 5200.28 (Series) Information Processing Standards for Computers (IPSC) Program. This instruction establishes the IPSC program for the Department of the Navy. It provides the framework for the development and implementation of ADP standards within the Department of the Navy and it provides the basis for formal Navy support of the international, national, Federal, and DOD levels of ADP standards development. The Department of the Navy IPSC program is an integral part of the Department of Defense IPSC program.

One of the policies of the Department of Defense IPSC program requires compliance with the Federal Information Processing Standards (FIPS) and American National Standards Institute (ANSI) as implemented through DOD instructions issued by the Assistant Secretary of Defense (Comptroller). One of the policies of the Department of the Navy's IPSC program is to establish standards within the Department of the Navy that may expand upon or abstract from, but will not conflict with Department of Defense, Federal, or American National standards.

1.1 FORTRAN PROGRAMMING

FORTRAN is a general-purpose, procedure-oriented, high-level language. It is largely computer independent and is designed primarily for programming scientific and engineering applications. This has resulted in the misconception that a detailed knowledge of mathematics is required in order to understand the language. Learning the syntax of FORTRAN is relatively simple; for instance, it is much less "wordy" than COBOL. The completion of this course is not intended to produce a polished programmer, but someone who is familiar with the rules of the language and its capabilities and, who, by consulting the appropriate references, could develop a moderately complex FORTRAN program.

Writing a program is not a single, simple task. It is best described as a series of steps. These steps are analysis (defining the problem), program flowcharting, program coding, program testing, and preparation for production (documentation).

It is of the utmost importance that the programmer clearly understand the program objective before attempting to write program statements. Nothing is more frustrating or less productive for a programmer than to complete a program and find that what was written is not what was needed.

Analysis involves a complete study of all the computational methods to be used, the source of data input, and the desired form of output. The actual design of the program follows the analysis of the program objective. A program flowchart of varying detail is constructed according to the complexity of the program. A program flowchart is a graphic display of the step-by-step computer processing required to accomplish the desired objective. A discussion of flowcharting symbols, flowcharting techniques, and programming terminology can be found in *Data Processing Technician 3 & 2*, NAVEDTRA 10264-D.

The actual coding of the program follows satisfactory completion of the program flowchart. Before the flowchart symbols are translated into program statements, the input and output records are defined. Arrays and various known switches and accumulators are also defined with names and initial values. After this "housekeeping" is completed, the programmer is concerned with following the logical paths provided in the flowchart as the program is coded.

Upon completion of the coding of the program, the next step is the testing or "debugging" step. It is quite common for the first few attempts to be unsuccessful. Usually the programmer has made mistakes such as simple coding errors, misunderstanding the format of the statements to be used, or keying errors when the statements were punched into cards or transcribed onto some other media. The compiler program, when run on the computer, provides a diagnostic listing along with a listing of the source program. The programmer then must correct the errors and repeat the compile process until all the language errors are eliminated. Successful elimination of language (syntax) errors does not guarantee that the program will produce correct results. The only sure way to prove the correctness of the logical design of the program is to actually run the program using data that represents all possible conditions that may be encountered. Regardless of the care that is exercised, it is normal to detect a flaw in a well-tested program even after a long period of operational use. Still, it is the duty of the programmer to make every effort to minimize this possibility.

The final step in the writing of a program is the preparation for production. Normally, documentation for two different uses must be prepared by the programmer.

First, instructions for operation of the program must be provided for the computer operators. Included in this documentation is an explanation of the purpose of the program, the form of the input and output, and any control settings or operator responses that may be necessary.

The second set of documentation should be thorough enough to enable programmers to understand the purpose of the program and the logic involved, thus enabling modification with minimum programming effort.

The five basic steps of writing a program (analysis, program flowcharting, program coding, program testing, and preparation for production), should always be followed by all programmers, whether they are beginners or veterans.

1.1.1 INFORMATION ABOUT THE COURSE

The figures that are included in this text are somewhat different than those you may have encountered in other courses. These figures contain examples, illustrations, and supplementary text. Many clarifying points are made in the figures.

There are exercises at the end of each chapter. These questions and their answers can be and should be used to reinforce the information presented in the body of the chapter. It is strongly recommended that these exercises be completed before starting the correspondence course for that chapter. The knowledge and practice gained in the exercises will aid in completing the correspondence course. In later chapters, the exercises require complete programs to be written. You may be unsure of how to begin; this is normal. Look at the solution given, get the idea, and then progress on your own. It is unlikely your solutions will be exactly like the given solutions. It is more important to understand how the given solution works.

It is the intent of the manual to adhere to the standards put forth in ANSI X3.9-1978. In some cases, only the minimum, generally used standard will be addressed in this text. Every possible choice for a given set of circumstances is not included.

A glossary of terms is included at the end of this manual as Appendix I.

Appendix II illustrates some of the steps in developing a FORTRAN program. A narrative coordinates the flowchart and program source statements. It is hoped that this program can be used throughout the course to expand on many points presented in the text.

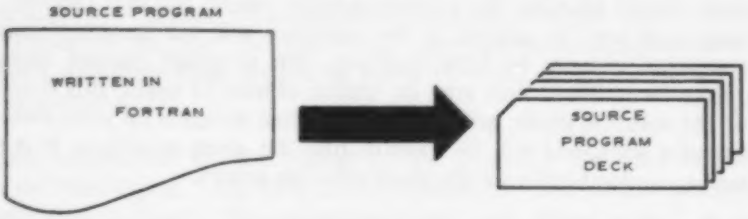
1.2 The FORTRAN Compiler

When writing a computer program in FORTRAN, the programmer describes what the computer is to do, in a language which closely approximates the algebraic notation. While such a language is very convenient for the programmer to understand and use, it is not so convenient for the computer itself and thus must be translated into machine language. This task of translating the FORTRAN source program into machine language is performed by the computer through the use of another computer program called a compiler. Each computer (IBM 4331, IBM 360, CDC 1604, Honeywell 6060, AN/UYK-7, etc.) requires a compiler for each higher level language (FORTRAN, COBOL, JOVIAL, etc.).

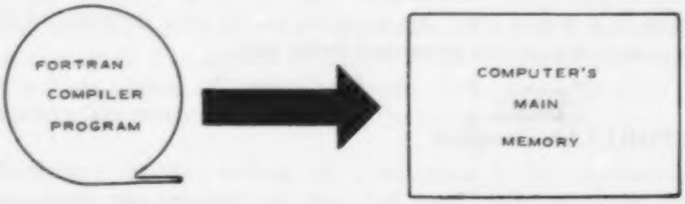
The FORTRAN compiler is designed so that it can translate all of the words and expressions allowed in the FORTRAN language, in accordance with its own rules of grammar, into the unique codes which comprise the machine language of the computer on which it is being used. Thus compilers must be designed for each type of computer in the industry, and the variation in design which results is one of the reasons why higher level languages such as FORTRAN are not completely machine independent. The compiler differences usually require some modification to a program written for one computer in order for it to be run on a different type of computer.

The compilation process consists of the following steps:

1. The FORTRAN source program is keypunched into cards to produce the source program deck.



2. The FORTRAN compiler program is entered into the computer's memory.



3. The source program deck (figure 1-1) is also read into the computer's main memory and control is turned over to the compiler. The machine operations to be performed are determined and the required machine language instructions are generated to form the object program. This translation process includes the assignment of memory locations for variables and constants, and the utilization of library routines and subroutines when required. The compiler produces an object program deck of machine instructions as well as a listing of the source program and a listing of syntax errors, if any. If, after testing, the program is determined to be satisfactory, the object program deck is stored for later use and the listing is saved for further reference should changes become necessary.

1.3 Compile and Run

A FORTRAN source program can be compiled and immediately executed. This is referred to as the Compile and Run process (figure 1-2). First the FORTRAN compiler is read into the computer's memory. The compiler program then reads the statements of the FORTRAN source program and translates them into an object program (machine language instructions) which is retained in the computer's memory. If the source program statements can all be translated, the object program is transferred to its execution memory locations and immediately run (data are read in and results are written out). This compile and run process gives the programmer the impression that the computer understood what was said in the FORTRAN language. It was developed to speed up the time taken to process a program written in a higher level language.

1.4 The FORTRAN Coding Form

FORTRAN source programs are written on a form similar to the one shown in figure 1-3. The information, normally one FORTRAN statement

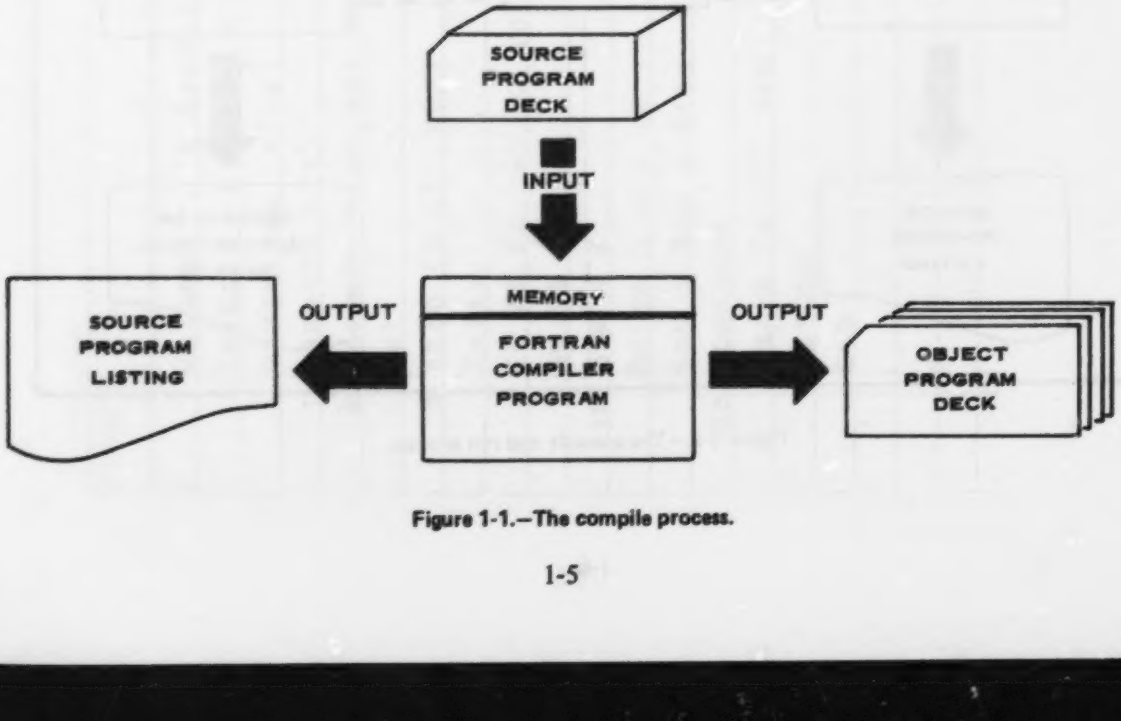


Figure 1-1.–The compile process.

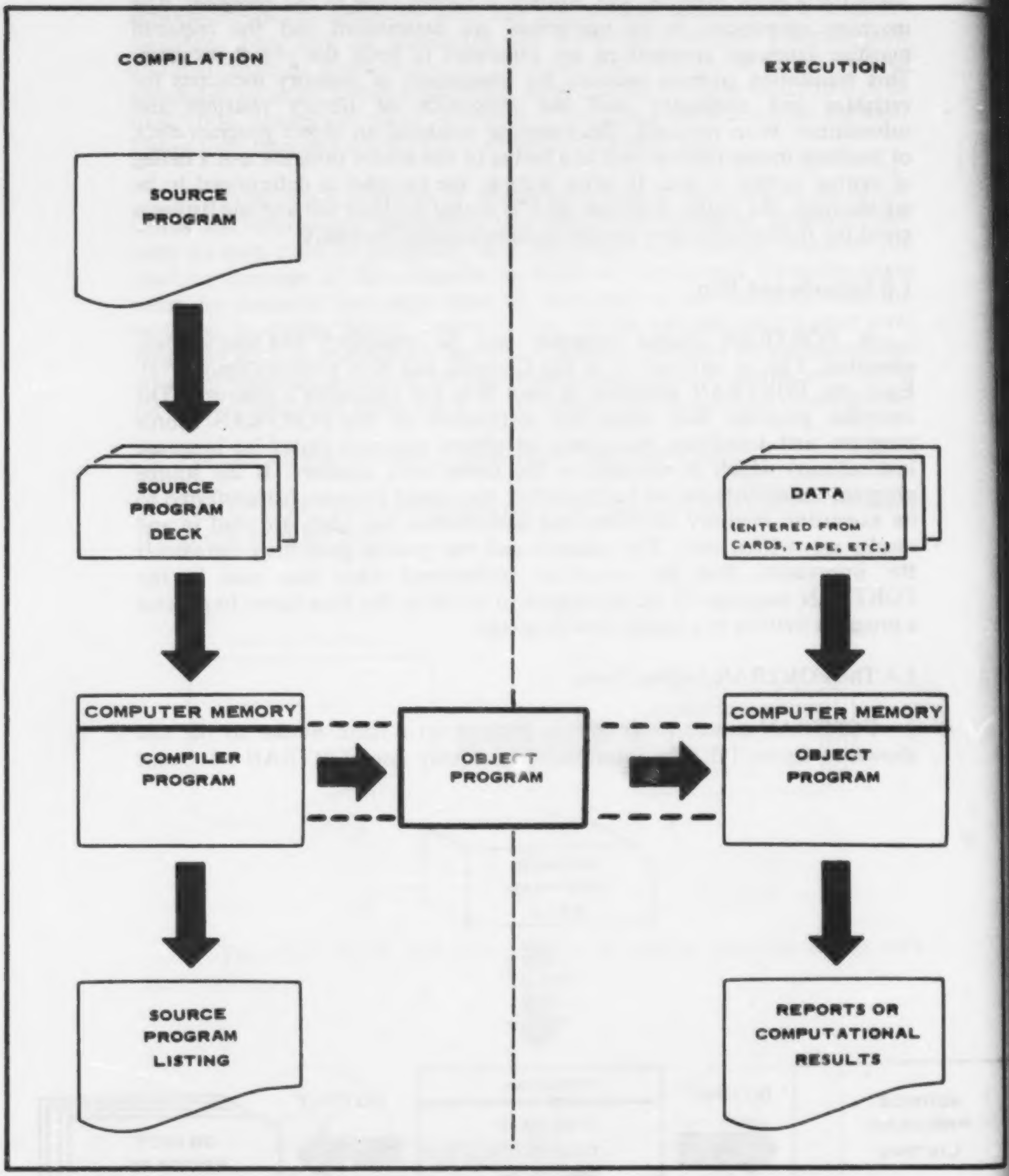
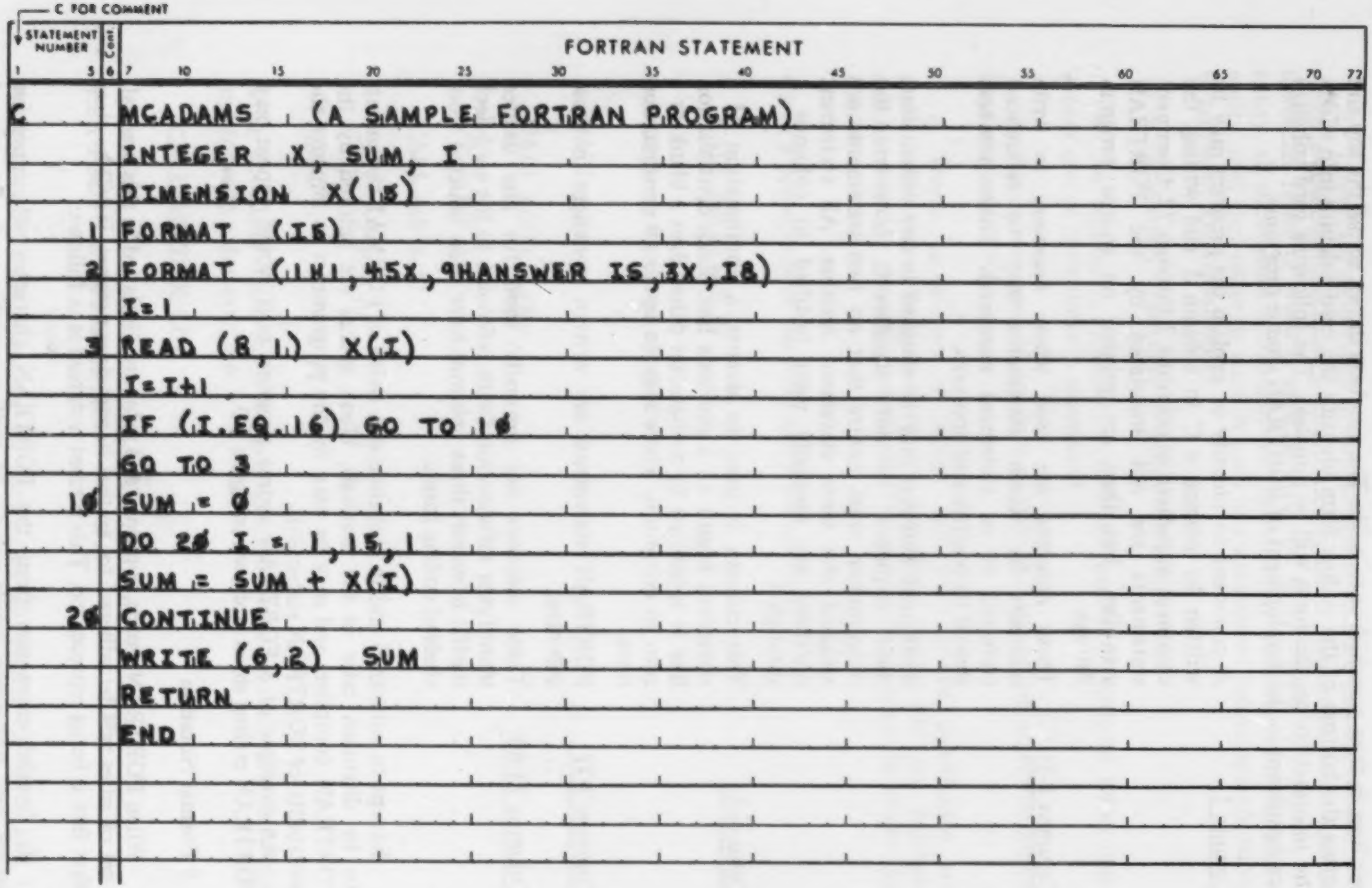


Figure 1-2.—The compile and run process.


C FOR COMMENT
STATEMENT NUMBER
Cont
FORTRAN STATEMENT
1
5
6
7
10
15
20
25
30
35
40
45
50
55
60
65
70
72

```
C     MCADAMS  (A SAMPLE FORTRAN PROGRAM)
      INTEGER X, SUM, I
      DIMENSION X(15)
    1 FORMAT (I5)
    2 FORMAT (1H1,45X,9HANSWER IS,3X,I8)
      I=1
    3 READ (8,1) X(I)
      I=I+1
      IF (I.EQ.16) GO TO 10
      GO TO 3
   10 SUM = 0
      DO 20 I = 1,15,1
      SUM = SUM + X(I)
   20 CONTINUE
      WRITE (6,2) SUM
      RETURN
      END
```

Figure 1-3.—FORTRAN statement.

per line, is then punched onto cards. The numbers shown across the top and across the bottom of the coding form identify the card columns into which the information on the form will be punched. The following card columns are significant in the development of FORTRAN source programs:

Column 1:	A comment to identify or explain the program may be written by placing a C in column 1 and writing the comment anywhere in columns 2 through 72. Comment statements are not translated by the FORTRAN compiler, but they are printed on source program listings.
Columns 1-5:	These columns are used where necessary to write numbers by which statements may be referenced (referred to as statement numbers). These numbers should be positive and non-zero. Statement numbers may be assigned in any order, since their sequence is not significant; however, the programmer must ensure that no two statements are assigned the same statement number. All statement numbers are normally right justified in columns 1 through 5.
Column 6:	This column is used to denote a continuation of a statement begun on a previous line. Each continuation line is identified by a character other than a blank or a zero in column 6. There may be up to 19 continuation lines.
Columns 7-72:	FORTRAN statements are written anywhere in these columns.
Columns 73-80:	These columns are normally used for any desired identifying information (with reference to the card deck itself); however, these columns have been omitted from student coding forms.

Except in column 6 and in alphanumeric fields of FORMAT statements (to be discussed later in the manual), blank spaces are ignored by the FORTRAN compiler, and may be used by the programmer to improve the readability of FORTRAN statements.

An example of a FORTRAN source program as it would appear on a FORTRAN coding sheet is shown in figure 1-3.

1.5 Format Notation

When FORTRAN words, symbols, and rules are displayed in this manual, a set of rules will be adhered to, so that in any circumstance, it will be clear what fact is being represented. This format notation is as follows:

1. Special characters from the FORTRAN character set, uppercase letters, and uppercase words are to be written exactly as shown, except where otherwise noted.

2. Lowercase letters and lowercase words indicate general entities for which specific entities must be substituted in actual statements. Once a given lowercase letter or word is used in a format specification to represent an entity, all subsequent occurrences of that letter or word represent the same entity until that letter or word is used in a subsequent format specification to represent a different entity.

3. Brackets, [ ], are used to indicate optional items.

4. An ellipsis, . . . , indicates that the preceding optional items may appear one or more times in succession.

5. Blanks are used to improve readability, but unless otherwise noted have no significance.

6. Words or groups of words that have special significance are underlined where their meaning is described. Titles and the symbols described in (2) are also underlined. The following example illustrates the format notation.

Given a description of the form of a statement as:

CALL sub [([a [, a] . . .])]

the following forms are allowed:

CALL sub

CALL sub ()

CALL sub (a)

CALL sub (a, a)

CALL sub (a, a, a)

etc

When an actual statement is written, specific entities are substituted for sub and for each a; for example:

CALL ABCD (X,1.0)

1.5.1 FORTRAN Character Set

The FORTRAN character set consists of 26 letters, 10 digits, and 13 special characters. A letter is one of the 26 characters A through Z, a digit is one of the 10 characters 0 through 9, and a special character is one of the 13 characters that follow:

	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal point
$	Currency symbol
'	Apostrophe
:	Colon

1.6 FORTRAN Statements

FORTRAN statements usually are constructed using certain FORTRAN key words with the basic elements of the language: constants, variables, and expressions. The categories of FORTRAN statements are as follows:

1. Assignment Statements: Cause calculations to be performed. The result replaces the current value of a specified variable or array element.

2. Control Statements: Enable the user to govern the sequence of execution of the object program and terminate its execution.

3. Input/Output Statements: Control input/output devices; enable the user to transfer data between internal storage and an input/output device.

4. FORMAT Statement: Used with certain input/output statements, specifies the form in which data appears in a FORTRAN record on an input/output device.

5. DATA Initialization Statement: Establishes the initial values of variables and array elements.

6. Specification Statements: Declare the properties of variables, arrays, and functions (such as type and amount of storage reserved).

7. Statement Function Definition Statement: Causes operations to be performed whenever the statement function name appears in an executable statement.

8. Subprogram Statements: Enable the user to name and to specify arguments for function and subroutine subprograms.

Figure 1-4 is a diagram of the required order of statements and comment lines for a program unit. Vertical lines delineate varieties of statements that may be interspersed. For example, FORMAT statements may be interspersed with statement function statements and executable statements. Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements. Note that an END statement is also an executable statement and must appear only as the last statement of a program unit. Figure 1-4 also combines categories 1 through 3 of Section 1.6 under the title of executable statements.

<table>
<tr><td rowspan="6">Comment Lines</td><td colspan="2">*PROGRAM, FUNCTION, or SUBROUTINE Statement</td></tr>
<tr><td rowspan="5">FORMAT Statements</td><td>*IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td>DATA Statements</td></tr>
<tr><td>*Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

*Not discussed in this elementary text

Figure 1-4.—Required order of statements and comment lines.

CHAPTER 1

EXERCISES

1. FORTRAN is an example of a __________ level programming language.

2. What is the general name of a program used to translate programs written in a higher level language to machine language?

   ____

3. FORTRAN was designed primarily for what purpose?

   ____

4. What is meant by a compile and run process?

   ____

   ____

   ____

5. a. What is a comment (as it applies to program coding)?

   ____

   ____

   ____

   b. Describe how a comment is entered on the FORTRAN coding sheet.

   ____

   ____

   ____

6. Are the following statements True or False?

   a. ___ Statement numbers can be entered anywhere in columns 1 through 5.

   b. ___ FORTRAN statements can be continued for more than one line.

   c. ___ Blank spaces are generally ignored by the FORTRAN compiler and thus may be used by the programmer to improve the readability of statements.

   d. ___ FORTRAN statements must begin in column 7 of the coding form.

   e. ___ Column 6 on the FORTRAN coding sheet is used to denote a comment.

CHAPTER 1

EXERCISES–Continued

7. When the format of a FORTRAN statement is presented, what method depicts an optional item?

8. What statement must appear last in a FORTRAN program?

CHAPTER 1

EXERCISE SOLUTIONS

1. Higher

2. Compiler program

3. Scientific and engineering applications

4. It is a process by which a FORTRAN source program can be compiled and immediately executed.

5. a. A comment can be used to identify or explain elements of the program.

   b. It is written by placing a C in column 1 and writing the comment in columns 2 through 72.

6. a. True (However they are normally right justified in column 5.)

   b. True

   c. True

   d. False (FORTRAN statements can be written anywhere in columns 7 through 72. However, statements generally begin in column 7.)

   e. False (Column 6 is used to denote line continuation.)

7. Brackets enclosing the optional item

8. The END statement

CHAPTER 2

CONSTANTS AND VARIABLES

2.1 FORTRAN Numbers

In FORTRAN, two types of numbers are generally used—integer and real. Each type is represented differently within the computer. Integer numbers (also referred to as fixed-point numbers) are ordinary whole numbers which are expressed without a decimal point; they may be zero or any positive or negative number. Examples of integer numbers are shown in figure 2-1.

Real numbers (also referred to as floating-point numbers) are numbers written with a decimal point. Examples of some real numbers are shown in figure 2-2. Note that while a fractional part is not required for a real number, the decimal point can never be omitted. The largest integer and real numbers permitted varies from computer to computer.

```
10000
  -35
 +150
 -150
```

Figure 2-1.—Valid integer numbers.

```
10000.564
    -35.9
  +150.48
  -150.48
      10.
```

Figure 2-2.—Valid real numbers.

2.2 Constants

Any number that appears in explicit form in an algebraic equation is called a constant. For example, in the following equation—

```
X = Y + 12.7
```

"12.7" is a constant; the "X" and "Y" are variables, which are discussed in the next section. It is possible to have both integer constants and real constants. FORTRAN distinguishes between the two types by the presence or absence of a decimal point.

The rules for using integer constants are:

1. They are written <u>without</u> a decimal point using the digits 0, 1, ..., 9.
2. They may be written with a preceding + or - sign. An unsigned constant is assumed to be positive.
3. The magnitude of the largest acceptable constant is specified by the computer being used. For work in this course, an integer constant must not be more than 11 decimal digits in length.

Some valid and invalid integer constants are shown in figure 2-3.

The rules for using real constants are:

1. They are written <u>with</u> a decimal point using digits 0, 1, ..., 9.
2. They may be written with a preceding + or - sign. An unsigned constant is assumed to be positive.

Valid Examples:

0
+9
186
-327

Invalid Examples:

-3.2	(contains a decimal point)
27,000	(contains a character other than a number or a sign)
27.	(contains a decimal point)

Figure 2-3.—Integer constants.

3. The magnitude of the largest, acceptable constant is specified by the computer being used. For work in this course, a real constant must not be more than eight decimal digits in length.

Some valid and invalid real constants are shown in figure 2-4.

2.3 Variables

A FORTRAN variable is any quantity which is referred to by a name (or alphabetic symbol) rather than a value. For example, in the arithmetic statement–

AREA = 3.14159*RADIUS*RADIUS

AREA and RADIUS are variables, while 3.14159 is a real (or floating-point) constant. Each time this equation is solved, RADIUS may have a different value. The value of AREA of course depends on the value of RADIUS.

Before a variable can be used in computations, it must be given a numeric value. There are two primary methods for doing this.

1. The value can be assigned by means of an input operation such as reading a data card. Thus, in the illustration above a value could be assigned to the variable RADIUS by reading a card which contained the required data.

2. The value can be assigned by means of an arithmetic operation. Therefore, AREA assumes a numeric value as a result of performing the calculation AREA = 3.14159*RADIUS*RADIUS, providing the variable RADIUS has been assigned a value.

Valid Examples:

1.
1.0
+1.04
-1.04

Invalid Examples:

4367 (no decimal point)
234483876. (exceeds number of digits allowed)

Figure 2-4.–Real constants.

The following rules govern the naming and use of variables:

1. The first character of any variable must be one of 26 letters of the alphabet.
2. All subsequent characters must be letters or numbers. No special characters such as +, -, *, /, or = are permitted in a variable name.
3. For convenience and interchangeability with other FORTRAN compilers, variable names should be no longer than six characters.
4. Like the numbers they represent, variables must be designated either integer or real. The first letter of the variable denotes the class to which it belongs. However, this rule can be overridden by the use of REAL and INTEGER types of statements, which are discussed in the next section.
    - a. A first character of I, J, K, L, M, or N denotes an integer or fixed-point variable. Numbers assigned to this variable must conform to the rules for integer numbers; i.e., be written without a decimal point.
    - b. A first character of any of the remaining letters of the alphabet denotes a real or floating-point variable. Numbers assigned to this variable must conform to the rules for floating-point numbers; i.e., be written with a decimal point.
5. Since the computer cannot differentiate between variables which have the same name, a specific name can be assigned only to one variable within the same program. For example, if more than one average is desired in a program, AVE, AVG, AVGE, AVG1 or other variations can be used to present the different averages.

Some examples of valid and invalid variable names are shown in figure 2-5.

Valid Examples:

Real --	TARGET X SUM
Integer --	I JOB1 NAME

Invalid Examples:

GS-15	(has special character)
8-PLACE	(does not begin with an alphabetic character; has special character)

Figure 2-5.—Variable names.

2.4 INTEGER and REAL statements

The use of INTEGER and REAL statements is another way in which integer and real variables can be denoted by the compiler. As already pointed out, this method overrides the first-letter convention introduced in Section 2.3. INTEGER and REAL statements fall under the general category of statements known in FORTRAN as Type Statements. Included in this category are INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and EXTERNAL declarations. Only the first two types are referred to in this manual.

The general form for REAL and INTEGER statements is–

REAL
or
INTEGER V_1 , V_2 , ...Vn

where–REAL or INTEGER	must appear as shown.
V_1, V_2, ... V_n	represent variable names; each variable must be separated by a comma, as shown.

The FORTRAN rules for REAL and INTEGER statements are:

1. The statements must appear at the beginning of the program prior to any usage of the variable in the program.

2. The words REAL and INTEGER are each written on a separate line on the coding form beginning in column 7.

The use of REAL and INTEGER statements is illustrated in figure 2-6.

Note that some of the variables included in the statements in figure 2-6 could have been omitted, and the compiler still would have been able to denote their type. Examples are X1 and TEAM, which are identified as REAL by the first letters X and T, respectively. The variables K and NAME are identified as INTEGER by their first letters. Although such inclusion in

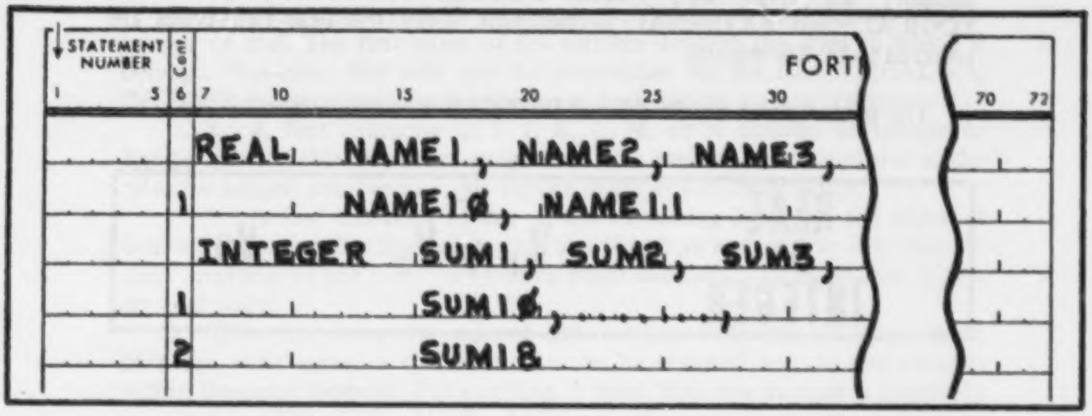
```
REAL      NUMBFD, X1, TEAM

INTEGER   RECORD, TOTAL1, K, NAME
```

Figure 2-6.–Use of REAL and INTEGER statements.

formal type statements might seem unnecessary, there is no harm done by such action.

More than one line of variables can be listed after either a REAL or INTEGER word by making use of the continuation column on subsequent lines. (See figure 2-7.) The number of continuation lines that can be used for each statement varies from computer to computer.


STATEMENT NUMBER
Cont.
FORT
1
5
6
7
10
15
20
25
30
70
72

```
      REAL  NAME1, NAME2, NAME3,
     1      NAME10, NAME11
      INTEGER SUM1, SUM2, SUM3,
     1        SUM10,.......,
     2        SUM18
```

Figure 2-7.—REAL and INTEGER statements.

CHAPTER 2

EXERCISES

1. For each variable listed, state whether or not the following are acceptable as integer variable names (INTEGER statement not used).

   a. AID
   b. AID,5
   c. M15M
   d. LEFT6
   e. K2345
   f. MARK-5
   g. ICAT
   h. 1JOKE
   i. RED SOX

2. Identify and correct any errors in each of the following floating-point constants.

   a. 23765
   b. 10.0
   c. 1,500.567
   d. -.95
   e. 0.

3. The following are names of integer and real variables. Some are acceptable and some are not. For each name given, determine whether it would be integer or real; if the name is unacceptable for either, explain why.

   a. AMOUNT
   b. VOLTS
   c. UNIT
   d. IVOLTS
   e. MATH
   f. AIIN8
   g. J72K4
   h. DEC1.0
   i. 92PUNT
   j. H20
   k. DEC-D
   l. K
   m. CO.F3

4. Variables can be given numeric values by means of ____________ or ____________ operations.

5. You must solve a problem in which the following computations must be made:

   Variable (3)=Variable (1) + Variable (2) - 42.6 (All data is real)
   Total Yards = Yards Rushing + Yards Passing (All data is integer)
   Provide names for the variables so that the appropriate number types are used in the computations.

   ____________ = ____________ + ____________ - 42.6

   ____________ = ____________ + ____________

CHAPTER 2

EXERCISE SOLUTIONS

1. a. No (incorrect first letter)
   b. No (incorrect first letter) and comma
   c. Yes
   d. Yes
   e. Yes
   f. No (has special character)
   g. Yes
   h. No (incorrect first character)
   i. No (blank not permitted; incorrect first character)

2. a. No decimal 23765.
   b. Correct
   c. Comma not permitted 1500.567
   d. Correct
   e. Correct

3. a. Real
   b. Real
   c. Real
   d. Integer
   e. Integer
   f. Real
   g. Integer
   h. Unacceptable (has special character)
   i. Unacceptable (incorrect first character)
   j. Real
   k. Unacceptable (has special character)
   l. Integer
   m. Unacceptable (has special character)

4. Input; arithmetic

5. One correct solution might be:

   Z = X + Y - 42.6

   TOTYDS = YDRUSH + YDPASS (with appropriate INTEGER statement)

CHAPTER 3

ARITHMETIC

3.1 Integer Arithmetic

In FORTRAN, integer calculations are made with integer numbers only; no decimal remainders are retained or used in subsequent computations. Results of performing integer arithmetic are illustrated in figure 3-1. Note particularly that the use of integer arithmetic to perform division is limited because of truncation.

3.2 Real (Floating-Point) Arithmetic

In FORTRAN, real calculations are made with real numbers only. (A real number can have an integer exponent.) Rounding does not occur; any excessive digits in the fractional portion of an answer are simply dropped (truncated). Some typical calculations in the real mode are shown in figure 3-2.

Arithmetic Statements	Results of Calculations
I = 5/2	I = 2 (instead of 2.5 since the .5 is dropped or truncated)
J = 5/2 + 7/2	J = 5 (intermediate truncation causes this to be computed as 2 + 3 rather than 12/2)
K = 5*2	K = 10 (the * signifies multiplication in FORTRAN)
L = -4 + 1	L = -3

Figure 3-1.–Examples of integer arithmetic.

Arithmetic Statements	Results of Calculations
A = .4301 / 1.7	A = .253
B = 5./2.	B = 2.5
C = 1.6*7.	C = 11.2
D = 2.7-1.2	D = 1.5

Figure 3-2.--Examples of real arithmetic.

3.3 Arithmetic Operations

There are five basic arithmetic operations in FORTRAN: addition, subtraction, multiplication, division, and exponentiation. The symbols which represent these operations are indicated in figure 3-3.

3.4 Arithmetic Expressions

An arithmetic expression is used to express a numeric computation. Evaluation of an arithmetic expression produces a numeric value.

A FORTRAN arithmetic expression can consist of a single constant, a variable, or a function (FORTRAN functions are covered in Section 3.6), or it can consist of two or more of these elements combined by the use of

Arithmetic Operation	FORTRAN Symbol	Example
Addition	+	A+B
Subtraction	-	199.5-C
Multiplication	*	AMPS*OHMS
Division	/	PI/2.0
Exponentiation	**	RADIUS**2.0

(Note that ** is considered as one symbol.)

Figure 3-3.--FORTRAN operation symbols.

operation symbols and parentheses to form more complicated expressions. Some examples of FORTRAN arithmetic expressions and their meaning are given in figure 3-4.

When writing expressions, a programmer must carefully observe certain rules in order to convey his/her intentions accurately to the FORTRAN compiler.

RULE 1: Parentheses greatly facilitate the writing of an arithmetic expression in a clear manner and should be used liberally. When an expression is enclosed in parenthesis, that expression is evaluated first. Two such examples are:

a. The product of the sum of A + B and the sum of C + D may clearly be written as (A + B)*(C + D); the expression A + B*C + D is not so clear and does not give the same results.

b. The quotient of A divided by B, multiplied by C is clearly expressed as (A/B)*C, whereas the statement A/B*C is not as clear.

RULE 2: No two arithmetic operation symbols can appear in sequence. Parentheses must be used to write expressions where two operation symbols would otherwise appear consecutively. Some examples of this rule are shown in figure 3-5.

Expressions	Explanations
A + 2.0	The sum of the value of A and 2.0
X*Y	The product of the values of X and Y
C**3	The value of C raised to the third power
565	The value of the integer constant 565

Figure 3-4.—Examples of arithmetic expressions.

Incorrect	Correct
X*-Y	X*(-Y)
(A*B)(-C^2)	(A*B)*(-C**2)

Figure 3-5.—Arithmetic expressions.

NOTE: Multiplication cannot be implied by parentheses. For example, (A + B)(C + D) must be written as (A + B)*(C + D).

RULE 3: When the order of operations in an expression is not completely specified by the use of parentheses, the order is interpreted (from left to right) by the computer as follows:

a. First, all exponentiations. Thus, for the expression,

A/B*C**2+D*E-F**3

the computer would first calculate C^2 and F^3.

b. Secondly, multiplications and divisions in a left to right order. Thus, in the foregoing expression, the computer would divide A by B and then multiply the quotient by C^2; then it would multiply D by E.

c. Finally, additions and subtractions in a left to right order.

Following this convention, the computer would interpret the expression A*X**2+B*X+C as follows: First it would generate the machine language instructions to perform the exponentiation, X^2; secondly, it would generate the instructions to multiply A times X^2 and B times X; and finally, it would generate the instructions to sum the three quantities $(AX^2) + (BX) + (C)$.

RULE 4: More than one set of parentheses may appear in an arithmetic expression. When the parentheses occur as separate sets, they are executed in a left to right sequence. In the expression

(A+B) - (C*D)/(E+X)

the A+B is calculated, then C*D is calculated, and then E+X is calculated. Finally, the product of C times D is divided by the sum of E plus X and that quotient is subtracted from the sum of A plus B. It is permissible to imbed one or more sets of parentheses within another set of parentheses. Evaluation of this type of expression begins with the innermost set of parentheses and progresses outward until all operations are performed. In the example

Y/((A-B)*C)

the innermost set of parentheses causes A-B to be calculated first. Then that remainder is multiplied by C and finally Y is divided by the value from the expression enclosed by the outer set of parentheses.

RULE 5: Parentheses must be used to raise the sum of two or more variables to a power. For example, the statement X+Y**2 would be interpreted by the compiler as $X + Y^2$, rather than $(X + Y)^2$. The statement should be written (X + Y)**2.

Type and Interpretation of Results for $X_1 + X_2$

X_2 X_1	I_2	R_2
I_1	$I = I_1 + I_2$	$R = REAL(I_1) + R_2$
R_1	$R = R_1 + REAL(I_2)$	$R = R_1 + R_2$

In the table above, + may be replaced by the symbols -, *, or /.

Type and Interpretation of Result for $X_1 ** X_2$

X_2 X_1	I_2	R_2
I_1	$I = I_1 ** I_2$	$R = REAL(I_1) ** R_2$
R_1	$R = R_1 ** I_2$	$R = R_1 ** R_2$

In both tables, REAL indicates the conversion from integer to floating-point.

Figure 3-6.—Mixed-mode arithmetic.

RULE 6: Integer and real quantities can be mixed in an expression with some restrictions. The type of the result, integer or real, of an expression is illustrated in figure 3-6. Except for a value raised to an integer power, figure 3-6 specified that if two different types of operands are used, the operand that differs in type from the result of the operation is converted to the same type as the result. The operator is then able to work with a pair of operands of the same type. When a type REAL is raised to an integer power, the integer operand need not be converted. Figure 3-7 contains examples of mixed-mode expressions. It is best not to mix modes.

A=A+B	No conversion is necessary, A and B are both Real variables.
A=A*J	J is an Integer variable, A is a Real variable. J would be internally converted to a Real value, and then floating-point multiplication would take place. This is an example from the Figure referenced in Rule 6. $R=R+REAL(I_2)$ demonstrates the expression $R=R=I_2$, where I_2 must be converted to Real.

Figure 3-7.—Mixed-mode expressions.

3.5 Arithmetic Statements

A FORTRAN arithmetic statement is a command to the computer to evaluate an arithmetic expression and then to replace the current value of some variable with the numeric result obtained. The general form of these statements is–

V = E

where–

V is a FORTRAN variable that is assigned a value as a result of evaluating E.

E is any FORTRAN arithmetic expression.

The programmer is not allowed to write statements such as X - 3 = Y + 4. The only legitimate form of a FORTRAN arithmetic statement is one in which the left side of the statement is a single variable, as indicated in the general form in the preceding example.

It is important to realize that in a FORTRAN arithmetic statement, the equal sign means "is to be replaced by" rather than "is equivalent to." Based on this meaning of the equal sign, a statement such as COUNT = COUNT + 1 means replace the value of the variable COUNT with the sum of its old value and 1. This sort of statement, which is clearly not a valid mathematical equation, finds frequent use in programming. One such use is the accumulation of tallies. Some examples of FORTRAN arithmetic statements are given in figure 3-8.

Statements	Explanations
A=3.0	Store the value 3.0 as A
B=2.0	Store the value 2.0 as B
C=A+B	Add the values of A and B and store the results as C

Figure 3-8.–Examples of FORTRAN arithmetic statements.

3.6 Functions and Subroutines

Sometimes it is desirable to write a program which, at different places, requires the same processing to be performed with different data for each instance. It simplifies the writing of that program when the statements required to perform the desired computation can be written one time, but can be used repeatedly. Each reference has the same effect as if these instructions were written at the point in the program where the reference was made.

For example, to calculate the area of a circle, a program must be written for this purpose. If a general-purpose program were written to calculate the area of a circle, it would be desirable to be able to combine that program (or subprogram) with other programs where these calculations are required.

The FORTRAN language provides two classes of subprograms: FUNCTION subprograms and SUBROUTINE subprograms. Also, there is a group of FORTRAN-supplied subprograms. The primary difference between FUNCTION and SUBROUTINE subprograms is that FUNCTION subprograms must return at least one value to the calling program; SUBROUTINE subprograms do not.

3.6.1 FORTRAN-Supplied Subprograms

FORTRAN provides a number of trigonometic and other mathematical functions which may be used in arithmetic expressions as if they were variables. The general form of a function expression is–

$$\boxed{f(e)}$$

where–

f is one of the preassigned function names. The exact list of functions which are available depends upon the compiler being used. Some of the more common functions are shown in figure 3-9.

e is the FORTRAN expression to be evaluated. (Note: The elements enclosed in parentheses are also referred to as the "arguments of the function.")

The user of functions must be aware of the type (REAL or INTEGER) restrictions of the arguments. This is clearly specified for each compiler. For example, any arguments used with the functions shown in figure 3-9, must be real quantities and the functional value is computed in real form. A complete list of the FORTRAN-supplied subprograms is available in ANSI X3.9-1978 or in the documentation of the specific compiler used.

Mathematical Function	Definition	FORTRAN Name
Sine	sin(x)	SIN
Cosine	cosine(x)	COS
Square Root	$\sqrt{x}$	SQRT
Exponential	e^x	EXP
Natural Logarithm	$\log_e(x)$	ALOG
Common Logarithm	$\log_{10}(x)$	ALOG10
Absolute Value	$\|x\|$	ABS

Figure 3-9.—Some common FORTRAN functions.

In order to use a FORTRAN function (referred to in most texts and manuals as library or built-in functions), it is necessary only to write the function in the form indicated. This is sufficient to cause the computer to perform those operations necessary to evaluate the named function using the expression in parentheses as its argument. The use of functions is illustrated in figure 3-10.

Example 1: Develop the square root of A and place the answer in B.

Solution: B=SQRT(A)

Example 2: Evaluate the mathematical expression–

$$\text{Discriminant} = \sqrt{b^2\text{-}4ac}$$

Solution: DISCR=SQRT(B**2-4.0*A*C)

Example 3: Multiply the absolute value of the sine of X by e^x and place the result in ANSW.

Solution: ANSW=EXP(X)*ABS(SIN(X))

Figure 3-10.—Use of FORTRAN functions.

3.6.2 User-Written Subprograms

A programmer who writes subprograms must be aware of several rules and apply them in the construction of subprograms.

A subprogram name follows the rules for a variable name. It consists of from one to six alphanumeric characters, the first of which must be alphabetic. A subprogram name must not contain special characters. The type (INTEGER or REAL) of the subprogram determines the type of the result that can be returned from it. The type may be determined by a predefined convention (see 2.3(4)) or through the use of an explicit specification in the FUNCTION statement, as shown in figure 3-11. The function name must also be TYPED in the program units which refer to it, if the predefined convention is not used.

3.6.2.1 FUNCTION Subprograms

A FUNCTION subprogram is a complete FORTRAN program consisting of a FUNCTION statement followed by other statements, including at least one RETURN statement. It is a program that is executed wherever its name is referred to by another program. See figure 3-11 for the format of a FUNCTION statement.

Because the FUNCTION is a separate program, it doesn't matter if the variable names and statement numbers within it are the same as those in other programs.

The FUNCTION statement is always the first statement in the subprogram. The FUNCTION subprogram may contain any other FORTRAN statement except a SUBROUTINE statement or another FUNCTION statement.

[Type] FUNCTION name (a_1, a_2, a_3, . . . a_n)

Where: Type is INTEGER or REAL: Its use is optional.
name is the name of the subprogram. (See 3.6.2)

If the predefined type convention,–i.e., a name beginning with I, J, K, L, M, or N is INTEGER otherwise the name is REAL,–is not sufficient, then the type may be specified by including INTEGER or REAL on the FUNCTION statement itself.

Each a is a dummy argument. It must be a unique variable or array name or dummy name of a SUBROUTINE or other FUNCTION subprogram. There must be at least one argument in the argument list enclosed in parentheses.

Figure 3-11.–FUNCTION statement.

To establish a value to be returned to the calling program, the name of the function must be assigned a value at least once during execution of the subprogram—as the variable name to the left of the equal sign in an arithmetic or a logical assignment statement or in a list of a READ statement within the subprogram. The READ statement is covered in Section 4.2.

Although the arguments on the FUNCTION statement are primarily used to pass values to the FUNCTION subprogram, after their values are utilized by the FUNCTION subprogram, one or more of the arguments may be used to return values to the calling program. Any argument so used will have values assigned to it in the same manner as the FUNCTION subprogram. It must appear on the left side of an arithmetic or a logical assignment statement or in the list of a READ statement within the subprogram.

The dummy arguments of a subprogram must appear after the FUNCTION name and are enclosed in parentheses. (See figure 3-11.) They are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. The dummy arguments must match the actual arguments in number, sequence, and type. For example, if an actual argument is an integer constant, then the corresponding dummy argument must be an integer. If a dummy argument is an array, the corresponding actual arguments must be either an array name, or an array element. For an array name, the size of the dummy array must not exceed the size of the actual array. For an array element, the size of the dummy array must NOT exceed the size of that portion of the actual array which follows and includes the designated element.

The actual arguments used during execution can be any of the following, as needed:

- The name of a FUNCTION or SUBROUTINE subprogram
- Any type of array name
- Any type of variable or array element
- A literal or arithmetic constant
- Any type of arithmetic expression

The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the FUNCTION subprogram is illustrated in figure 3-12.

In example 1 of figure 3-12, the values of X, Y, and I are used in the FUNCTION subprogram as the values of A, B, and J, respectively. After the value is computed and placed in CALC, this value is returned to the calling program where the value of ANS is computed. The variable I in the argument list of CALC in the calling program does not conflict with the variable I

Example 1:

Calling Program	FUNCTION Subprogram	Actual Arithmetic Operation
.		
.	FUNCTION CALC (A,B,J)	
.	.	
ANS = ROOT1*CALC(X,Y,I)	.	
.	.	
.	I = J*2	I = 2 X 2 (4)
.	.	
.	CALC = A**I/B	CALC = $\frac{5^4}{2}(312.5)$
	RETURN	
	END	

Example 2:

Calling Program	FUNCTION Subprogram
INTEGER CALC	INTEGER FUNCTION CALC (I,J,K)
.	.
.	.
.	
ANS=ROOT1*CALC(L,M,N)	CALC=I+J+K**2
.	.
.	.
	RETURN
	END

Figure 3-12.–FUNCTION subprogram examples.

appearing in the subprogram. In this example, if X, Y, and I were given the values 5, 2, and 2, respectively, the FUNCTION subprogram CALC would return the value 312.5 to the calling program to be used in the computation of ANS. (Note: do not forget the hierarchy of arithmetic operators.)

In figure 3-12, example 2, the subprogram CALC has been declared as type INTEGER.

3.6.2.1.1 RETURN and END Statements in a FUNCTION Subprogram

All FUNCTION subprograms must contain at least one RETURN statement and have as its last statement an END statement. The END statement signifies the physical end of the subprogram; the RETURN statement denotes a logical conclusion of the computation and returns the computed function value and control to the calling program. RETURN statements and END statements are discussed in paragraphs 5.9 and 5.10.

3.6.2.2 SUBROUTINE Subprograms

The SUBROUTINE subprogram (figure 3-13) is similar to the FUNCTION subprogram in many ways. The rules for naming FUNCTION and SUBROUTINE subprograms are the same. They both require an END statement, and they both may contain the same sort of dummy arguments. Like the FUNCTION subprogram, the SUBROUTINE subprogram is a set of commonly used computations, but it does not have to return any results to the calling program, like the FUNCTION subprogram. The SUBROUTINE subprogram is invoked in a calling program by the CALL statement.

Since the SUBROUTINE is a separate program, it does not matter if the variable names and statement numbers within it are the same as those in other program units.

The SUBROUTINE subprogram must begin with a SUBROUTINE statement. The SUBROUTINE subprogram may contain any other FORTRAN statement except a FUNCTION statement or another SUBROUTINE statement.

Although the arguments on the SUBROUTINE statement are used primarily to pass values to the SUBROUTINE subprogram, after their values are utilized by the SUBROUTINE subprogram, the SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. An argument used in this manner appears on the left side of an arithmetic or logical assignment statement or, in the list of a READ statement within the subprogram. The SUBROUTINE name may not appear in any other statement in the SUBROUTINE subprogram.

The dummy arguments ($\underline{a}_1$, $\underline{a}_2$, $\underline{a}_3$, . . . , $\underline{a}_n$) on the SUBROUTINE statement are dummy names that are replaced at the time of execution by the actual arguments supplied in the CALL statement.

SUBROUTINE $\underline{\text{name}}$ [(a_1, a_2, a_3, . . . a_n)]

Where: $\underline{\text{name}}$ is the SUBROUTINE name. (See 3.6.2)

Each $\underline{a}$ is a unique dummy argument (i.e., it may appear only once within the statement). The arguments are optional, therefore the parentheses must be omitted. Each dummy argument used must be a variable or array name or another SUBROUTINE or FUNCTION subprogram name.

Figure 3-13.—SUBROUTINE statement.

The dummy arguments of a subprogram must appear after the SUBROUTINE name and are enclosed in parentheses. They are replaced by the actual arguments supplied in the CALL statement in the calling program. The dummy arguments must match the actual arguments in number, sequence, and type. For example, if an actual argument is an integer constant, then the matching dummy argument must be an integer. If a dummy argument is an array, the corresponding actual argument must be either an array name, or an array element. For an array name, the size of the dummy array must not exceed the size of the actual array. For an array element, the size of the dummy array must not exceed the size of that portion of the actual array which follows and includes the designated element.

The actual arguments used during execution can be any of the following as needed:

- The name of a FUNCTION or SUBROUTINE subprogram
- A literal or arithmetic constant
- Any type of arithmetic expression
- Any type of variable or array element
- Any type of array name

When control returns from a SUBROUTINE subprogram to the calling program, all variables and arrays in the subprogram that are not in common and are not dummy arguments become undefined, except those given initial values in a DATA statement or whose initial values were not changed.

The CALL statement (figure 3-14) is used to call or invoke a SUBROUTINE subprogram. The CALL statement transfers control from a calling program to the SUBROUTINE subprogram and replaces the dummy

CALL name [(a_1, a_2, a_3, . . . a_n)]

Where: name is the name of a SUBROUTINE subprogram.

Each a is an actual argument in the calling program that is available to the SUBROUTINE subprogram. The argument may be a variable, array element, or array name, a literal, an arithmetic expression, or a function or a subroutine name.

Figure 3-14.–CALL statement.

variables with the values of the actual arguments that appear in the CALL statement.

Control normally returns to the statement in the calling program that follows the CALL statement. (See figure 3-15).

Example:

Calling program

```
10 CALL RADIAN (THETA)
20 X=THETA
          .
          .
          .
```

Subroutine Subprogram

```
SUBROUTINE RADIAN (ALPHA)
          .
RAD=ALPHA*3.1416/180
ALPHA=RAD
          .
          .
          .
          .
RETURN
END
```

The Subprogram RADIAN computes RAD using the THETA argument value. The answer developed is returned to the calling program.

Figure 3-15.—SUBROUTINE subprogram example.

CHAPTER 3

EXERCISES

1. Give the FORTRAN symbol and write an example for each of the following arithmetic operations.

Operation	FORTRAN Symbol	Example
a. Multiplication		
b. Subtraction		
c. Division		
d. Exponentiation		
e. Addition		

2. a. List each step performed by the computer in solving the following equation. A=B+C/D**2*(F-2.0)

Step 1:

Step 2:

Step 3:

Step 4:

b. Calculate the value at each step for:

B = 15.4
C = 32.16
D = 4.0
F = 8.0

Step 1:

Step 2:

Step 3:

Step 4:

A =

CHAPTER 3

EXERCISES–Continued

3. Write FORTRAN expressions for each of the following mathematical expressions.

Mathematical Expression	FORTRAN Expression
a. $a \times b$	
b. $a + 3$	
c. $-(a+b-c)$	
d. b^2	
e. $\frac{a \times b}{c \times d}$	
f. $(a \times b)^{x+3}$	
g. $1 + \frac{a}{10-x}$	
h. $x^2 - 500$	

4. Determine the value of A or I, according to FORTRAN rules for arithmetical expressions, for each of the following:

a. A = 2.0*4.0+1.0 A = ____________

b. I = 14/5*2 + 3 I = ____________

c. A = 14.0/5.0*2.0 + 3.0 A = ____________

d. A = 2.0+3.0*4.0+6.0 A = ____________

e. A = (2.0+3.0)*4.0+6.0 A = ____________

f. A = (2.0+3.0)*(4.0+6.0) A = ____________

g. I = (2**3)**2 I = ____________

CHAPTER 3

EXERCISES–Continued

5. Point out the errors if any, in each of the following FORTRAN expressions.

Mathematical Expression	FORTRAN Expression	Error
a. $\left(\frac{a}{b}\right)^{x+5}$	(A/B)**X+5	
b. a x b x c / c+2	ABC/C+2.0	
c. c · 2/ .95	(C*2)/.95	
d. -4ac· 5000	(-4)*A*C*5000	
e. $(m \cdot n)^{2x}$	M*N*2X	

6. Write arithmetic statements to compute the following formulas.

   a. $\text{Area} = \pi R^2$

   b. $\text{Gradient} = 1/2 \log_e \frac{1+\sin x}{1-\sin x}$

   c. $\text{Altitude} = \pi X \cos \frac{X}{A} - \frac{A}{2} \log_e (A^2+X^3)$

   d. $\text{Quad} = \frac{2}{\pi X}^{3/2} \cos X$

7. What three statements must all SUBROUTINE subprograms have?

8. Judge this to be TRUE or FALSE. The variable names in a subprogram must be different than any other variable name in the CALLING program?

CHAPTER 3

EXERCISE SOLUTIONS

1. a. *

b. -

c. /

d. **

e. +

2. a. Step 1: F - 2.0 is evaluated

Step 2: D^2 is evaluated

Step 3: C/D^2 and then C/D^2 times (F-2.0) are evaluated

Step 4: B is added to the result of Step 3

b. Step 1: (F-2.0) = 8.0 - 2.0 = 6.0

Step 2: $D^2 = (4.0)^2 = 16.0$

Step 3: C/D^2 * (F-2.0) = 32.16/16.0 = 2.01
then 2.01 X 6.0 = 12.06

Step 4: B + Result of Step 3 = 15.4 + 12.06 = 27.46

A = 27.46

3. a. A*B

b. A+3.0

c. -(A+B-C) or -A-B+C

d. B**2

e. (A*B)/(C*D)

f. (A*B)**(X+3.0)

g. 1.0+(A/(10.0-X))

h. (X**2)-500.0

CHAPTER 3

EXERCISE SOLUTIONS–Continued

4. a. A = 9.0
   b. I = 7
   c. A = 8.6
   d. A = 20.0
   e. A = 26.0
   f. A = 50.0
   g. I = 64

5. a. X+5 should be in parentheses.
   b. Missing operator symbols; implied multiplication is not permitted.
   c. Correct
   d. Correct (mixed-mode is permissible, but not advised)
   e. Missing operator between 2 and X; M*N should be placed in parentheses.

6. a. `AREA = 3.1415 * R ** 2`
   b. `GRAD = .5*ALOG((1.0+SIN(X))/(1.0-SIN(X)))`
   c. `ALT = 3.1415*X*COS(X/A)-(A/2.0)*ALOG ((A**2)+(X**3))`
   d. `QUAD = ((2/(3.1415*X))**1.5)*COS(X)`

7. SUBROUTINE, RETURN, END

8. FALSE

CHAPTER 4

INPUT/OUTPUT OPERATIONS

4.1 Introduction

Input statements cause the computer to read information from outside the computer (e.g., from a punched card) into the computer's memory. Output statements do the reverse. They transfer data from the computer's memory to some output media (e.g., a printed answer sheet or a punched card). The process of transferring data into a computer's memory is known as reading. The process of transferring data from the computer's memory to an output medium is known as writing.

The data that is external to the computer memory is located in a file or data set. File organization is divided into two basic types, sequential and direct access. Accordingly, there are FORTRAN input/output statements to access both types of organizations. Sequentially organized files have the following characteristics: the order of the records on the file is the order in which the records were written; the records must either all be created with FORMAT statements or all created without the use of FORMAT statements, except the last record may be an endfile record; and the records must not be read or written by input/output statements that are used for direct access files. A prime consideration of sequentially organized files is that in order to read any one specific record on the file, all records that precede the desired record must be read first.

A direct access file has the following characteristics. The order of the records is the order of their record numbers. The records may be read or written in any order. The records of the file are either all formatted or all unformatted. Reading and writing of records are accomplished only by direct access input/output statements. All records of the file must be the same length. Each record of the file is uniquely identified by a positive integer called the record number. The record number of a record is specified when the record is written. Once established, the record number of a record cannot be changed. Note that a record may not be deleted; however, a record may be rewritten. Records need not, but nevertheless, can be read or written in the order of their record numbers. Any record may be written into the file in any sequence. For example, it is permissible to write record 3, even though records 1 and 2 have not been written. Any record may be read from the file in any sequence, provided the record was written after the file was created.

It is the intent of this text to discuss only the FORTRAN input/output statements concerning sequentially organized files.

The following information must be specified for an input or output operation:

a. The input/output device to be used.

b. The data to be used.

c. The format or layout used for recording the data on punched cards, magnetic tape, or computer printouts.

In FORTRAN, a sequential I/O statement (e.g., READ, WRITE) is used to specify the operation to be performed. It is followed by a special number which specifies the device and a list of variables which specifies the data to be used. A separate format statement specifies the details of the layout of the physical record.

4.2 READ Statement

The READ statement is used by FORTRAN to enter data into the computer for processing. The general form of this statement is—

READ (X, n, ERR=s, END=s) a_1, a_2, a_3, ...a_n

where

READ	directs the transfer of data from an input device into memory.
X	is a digit which identifies the special input device being used. Device numbers normally used in FORTRAN are: 8—a magnetic tape drive 5—the card reader but these numbers can be changed on most systems.
n	refers to the statement number of the FORMAT statement which gives the format of the elements in a data record (i.e., integer or real quantity, number of digits required, etc.). FORMAT statements are covered in Section 4.4.

ERR=s	where s is the statement label of an executable statement that appears in the same program unit as the READ statement. If an error condition occurs during the execution of the READ statement, execution of the input/output statement stops and execution of the program continues with the statement labeled s. ERR=s is an optional parameter of the READ statement.
END=s	where s is the statement label of an executable statement that appears in the same program unit as the READ statement. When the READ statement encounters an endfile record and no error condition exists, execution of the READ statement stops and execution of the program continues with the statement labeled s. END=s is an optional parameter of the READ statement. END=s and ERR=s may appear in either sequence. See Section 4.2.1 for further information on endfile records.
$a_1, a_2, a_3, \ldots a_n$	are variable names of the data fields being read. They must be separated by a comma and must be in the same order as they appear in the data record.

Two examples of the READ statement are shown in figure 4-1.

Statement	Explanation
READ (8,2) RATE, TIME	The computer will read data from device number 8. The input format is FORMAT statement number 2. The two values read are the variables RATE and TIME.
READ (5,1,END=30,ERR=40)CL,NUM,GPA	The computer will read data from device number 5. The input format is FORMAT statement number 1. When the end of the file is reached, the execution of the program will continue at statement 30. If an error occurs while attempting to read the file, control of the program will be passed to statement 40. The values read are the variables CL, NUM, and GPA.

Figure 4-1.–The READ statement.

4.2.1 End-of-file Concepts

It is appropriate at this time to discuss the concept of sequential file organization in regard to the end-of-file (EOF) record. Consider a file containing an unknown number of records. How can a program determine when all of the records have been read? This is normally accomplished by the presence of a special record immediately following the last data record in a sequential file.

The most familiar type of end-of-file record is the delimiter card that follows a deck of data cards. This end-of-file record is generally unique to a particular computer system. Various IBM systems use the characters slash (/) and asterisk (*) in card columns 1 and 2 to serve as an EOF record; some BURROUGHS systems use ?END in card columns 1 through 4; and some UNIVAC systems use *EOF in card columns 1 through 4.

When an EOF record is read by a FORTRAN READ statement that has the END=s option included, control of the program is passed to the statement coded as s. At this point, other processing may be accomplished or the program may be terminated.

Since the END=s is optional to the FORTRAN READ statement and may not be available on some systems, other techniques are employed to determine when all data has been processed. A common method used by many FORTRAN programmers involves the inclusion, with the data, of an EOF record of the programmer's own choosing. The program then logically tests for its presence after every read. The obvious drawback of this plan is the creation of many different EOF records. Some systems automatically terminate execution of the program if the end of file is encountered and END=s has not been specified. Quite often there is a fixed number of records to be processed and a simple count of the records read indicates completion. Always check the individual circumstances for the best solution.

FORTRAN programs are also used to create files on magnetic media such as tape or disk. It is not possible to include one of the previously mentioned system delimiter cards with a file such as a magnetic tape. The techniques of a programmer-chosen delimiter or a known number of records is a possibility in some cases, but a method more frequently used involves using the FORTRAN-supplied statement ENDFILE. The format of the ENDFILE statement is–

```
ENDFILE u
```

where u is the device number associated with the file being created. When it is determined by the program that no more data needs to be written on the file being created (usually when the input is exhausted), a properly coded ENDFILE statement for the file is executed and a system end-of-file record is written immediately following the last data record. In future uses of the newly created file, the EOF record indicates the end of data and is compatible with the use of the END=s option of the FORTRAN READ statement.

Two other FORTRAN statements are commonly associated with the creation of sequential files. Their formats are–

```
BACKSPACE u
REWIND u
```

where

BACKSPACE u	causes the file referenced by the device number u to be backspaced one record for each execution of the BACKSPACE statement. It is normally used when it is desirable to reread a record.
REWIND u	causes a tape file referenced by the device number u to be rewound and positioned at the first record on the file. It is normally executed after the ENDFILE statement has been executed. This rewinds the tape to facilitate removal from the tape drive or ready the file to be further processed. For a disk file, the REWIND statement simply makes the first record the next record available for processing.

4.3 WRITE Statement

The WRITE statement is used in a program when data is to be written from the computer's main memory to an external device. The general form of this statement is–

$$\boxed{\mathrm{WRITE}\ (X,\ n)\ a_1,\ a_2,\ a_3,\ \ldots.a_n}$$

where

WRITE	directs the transfer of processing results from memory to an output device.
X	is a digit which identifies the output device being used. In the example, X=6 will be used; "6" refers to the line printer.

n	refers to the statement number of the FORMAT statement which gives the format in which the data is to be outputted.
$a_1, a_2, a_3, \ldots a_n$	represents the variables whose values are to be written out.

An example of the use of a WRITE statement is shown in figure 4-2.

4.4 FORMAT Statement

FORMAT statements describe data for input and output operations. The programmer can either place all of the required FORMAT statements in one part of the program or place each FORMAT statement near the READ or WRITE statement which refers to it. The general form of this statement is–

$$\boxed{\text{n FORMAT } (S_1,\ S_2,\ \ldots.S_n)}$$

where

n	is the statement number which must be referenced by the appropriate READ or WRITE statement.
FORMAT	must always appear as shown.
$s_1, s_2, \ldots s_n$	represent separate field specifications which describe the kind of information that is contained in input data records or in output print lines (blanks, integer or real numbers, alphanumeric characters, etc.) or other media.

There is a one-for-one relationship between each "s" and a field of the input record or output record. This means that the field sequence of the FORMAT statement must adhere to the sequence of fields on, for example, a card. Each field specification must be separated by a comma, and all of the specifications are enclosed in parentheses.

Statement	Explanation
WRITE (6,4) DIST, RATE, TIME	Values will be written out on the line printer (device number 6) in the format specified by FORMAT statement number 4. Values for the variables DIST, RATE, and TIME will be written.

Figure 4-2.–The WRITE statement.

The use of the FORMAT statement is illustrated in figure 4-3.

It is important also to note that during either input or output operations, the FORMAT statement is the determining factor as to how many variables are read or written. In either case, only the number of field specifications in the FORMAT and their corresponding variable names in the READ or WRITE statements are processed.

The following types of field specifications are examined in this manual: integer (I), real (F), Hollerith (H), blank (X), and alphanumeric (A).

a. Integer (I) Specification.

The general form of this specification is–

```
FORMAT (n I w)
```

where

n	is the number of integer values being described; if no value is given for n, the value is taken to be one. This is also called the repetition factor.
I	indicates an integer field.
w	is the total number of positions of the field.

The data card:

Actual Data →

999 (FIELD 1) Col 1-3	Blank Col 4-8	7056 (FIELD 2) Col 9-12	Blank Col 13-80

The READ Statement:

```
READ (5, 1) FIELD1, FIELD2
```

The FORMAT Statement:

```
1 FORMAT (I3, 5X, I4)
```

NOTE: This FORMAT statement describes two integer fields of 3 and 4 digits, separated by 5 unused positions.

Figure 4-3.–Use of the FORMAT statement.

If two or more specifications are given in a FORMAT statement, they must be separated by commas.

Some examples of I specifications are shown in figure 4-4.

If no sign is punched on a data card, the sign is assumed to be positive. Therefore, if a negative number is desired, the negative sign <u>must</u> be punched on the card immediately prior to the first digit of the value. The field size must include this position. For example, suppose the values 4671 and -6385 are to be read in as the values for the variables ITAX and LOSS. These would be punched as–

Card Column	1	2	3	4	5	6	7	8	9
Data	4	6	7	1	-	6	3	8	5

Now, suppose the following instructions are given–

```
10 FORMAT (2I4)
   READ (5, 10) ITAX,  LOSS
```

The result would be that 4671 would be assigned to ITAX and -63 would be assigned to LOSS, and this is NOT what was desired. Thus, the instructions for this example must be–

```
10 FORMAT (I4,  I5)
   READ (5,10)  ITAX,  LOSS
```

With these instructions, the desired values will be assigned to the variables ITAX and LOSS.

<u>Statements</u>	<u>Explanations</u>
1 FORMAT (I7) READ (5, 1) N	The value of N is an integer number with 7 or less digits.
2 FORMAT (I2, I7) WRITE (6, 2) N, M	This statement specifies an output of two integers per line. The first value, N, contains 2 or less digits; the second value, M, has 7 or less digits.
3 FORMAT (3I2, I5) READ (5, 3) I, J, K, L	The values for I, J, and K are integer numbers with 2 or less digits. The value of L is an integer number with 5 or less digits.

Figure 4-4.—Integer specification.

Note that on the reading of data cards, no blank spaces are required to separate different values. The field width portion of the specification controls the number of columns read for each value.

When writing output specifications for either integer (I) or real (F) type fields, a position for the sign should always be included in the field width. If the number of digits to be outputted exceeds the field width (w), the entire field will be filled with asterisks. However, if the number of digits to be outputted equals the field width (w), the negative sign will be lost and an incorrect value could be represented.

b. Real (F) Specification.

This specification is used for input and output of real numbers. The general form of this specification is–

nFw.d

where

n	is the number of real values being described; if no value is given for n, the value is taken to be one.
F	indicates a real field.
w	is the total number of digits in the field (including a plus or minus sign and the decimal point).
. (period)	must appear as shown.
d	is the number of digits to the right of the decimal point in the data field. (Note: If there is no fractional part, a zero must be used after the decimal in the specification.)

An example of F specification is shown in figure 4-5.

When reading F-type data, a value is considered positive unless the sign is given, and the position of the decimal point need not be punched. For example, consider that the following values are punched in the indicated columns–

Card Column	1	2	3	4	5	6	7	8	9	10	11	12	13
Data	-	7	6	5	4	3	2	1	0	9	8	7	6

The instructions

```
10  FORMAT (F7.2)
    READ (5,10) COST
```

result in the value -7654.32 being assigned to the variable COST.

Statements	Explanation
1 FORMAT (F10.4, F8.1) READ (5, 1) X, Y	The input consists of two real numbers per data card. The value for X is 10 digits in length (including a sign and decimal point) and has 4 digits to the right of the decimal point. The value of Y is 8 digits in length and has 1 digit to the right of the decimal point. An example of a valid number to fit the specification F10.4 would be +3345.6218, and an example to fit F8.1 would be -10425.9.

Figure 4-5.—Real specification.

The instructions

```
10  FORMAT (2F5.2)

    READ (5,10) COST, PRICE
```

result in the value -76.54 being assigned to COST and 321.09 being assigned to PRICE.

If the decimal point is punched on the card, it must be counted in the field width. It also overrides the d portion of the specification, if they differ. For example—

Card Column	1	2	3	4	5	6	7	8	9	10
Data	-	7	.	6	5	4	3	2	8	5

The instructions

```
10  FORMAT (F7.2)

    READ (5,10) CHARGE
```

result in the value -7.6543 being assigned to the variable CHARGE.

When writing format statements for output, the decimal point <u>must</u> be counted as a print position, and a position must be left for the sign, if one is to be printed. As examples of what can happen, assume that as a result of computation, the computer has assigned a value of -123.456381 to the variable DUD. If the instructions were

```
10  FORMAT (F6.3)

    WRITE (6,10) DUD
```

the value 23.456 would be printed. Note that the sign has been lost and the number is nowhere near the true value. However, if the format statement had been

```
10 FORMAT (F9.4)
```

then the value -123.4564 would have been printed. Note that a rounding operation takes place during the output of real numbers. Also remember that during output, if a number to be printed is positive, a plus sign will NOT be printed even if space is allowed. This is similar to the input consideration that a number is positive when there is no sign present.

c. Alphanumeric (H or Hollerith) Specification.

This specification is utilized by the programmer to WRITE output table and column headings or alphanumeric messages. The H specification is different from the I or the F specification in that it is not associated with any variable in a READ or WRITE statement. Instead, it calls for the output of a text containing alphanumeric characters. The general form of this specification is–

```
nHxxxxxxx.....x
```

where

n	designates the number of characters to be written; any character included in the FORTRAN character set, including blanks, is permitted.
H	indicates a Hollerith field.

The use of the H specification is illustrated in figure 4-6. It is extremely important that the count preceding the H in each specification be exact. Otherwise an error will result during program compilation.

Statements	Explanation
`1 FORMAT (9HTAX TABLE)` `WRITE (6, 1)`	The 9 refers to the number of characters to be written, H specifies a Hollerith message, and TAX TABLE is the message consisting of 9 characters (including the blank). When this WRITE instruction is executed, the message TAX TABLE will be written.

Figure 4-6.–H Specification.

A frequently used alternative to the Hollerith specification for character output is the use of the string of characters desired to be enclosed in single quotation marks or apostrophes. The length of the 'literal,' as the character string is known, is simply the number of characters within the apostrophes. An apostrophe within the literal must be represented by two consecutive apostrophes. For example, to cause MIKE'S to appear properly, the FORMAT statement would be coded as follows–

```
1 FORMAT ('MIKE''S')
```

The two apostrophes are recognized as one within the literal.

d. Blank (X) Specification.

This specification is used to separate data in input records and output lines with blanks for ease of reading. It is always used in conjunction with the other specifications. The general form is–

```
wX
```

where

w	specifies the number of blanks.
X	indicates a blank field.

The use of the blank specification is illustrated in figure 4-7.
The alphanumeric (A) specification is discussed in chapter 7.

Statements	Explanation
FORMAT (6X, I7)	This statement specifies a field of 6 blanks to the left of the integer number.
FORMAT (I2, 15X, I7)	There will be 15 blanks between the two integer numbers.

Figure 4-7.–Use of the blank specification.

4.5 Carriage Control

One of the most frequent applications of the Hollerith field specification is for controlling the spacing between print lines. The first character to the right of the H of a WRITE FORMAT statement is used to control this spacing (referred to as carriage control). If the carriage control character is a blank, normal single spacing occurs before a line is printed. If the character is a 0 (achieved by a 1H0 specification), double spacing occurs.

If the control character is a 1, the paper spaces to the top of the next page before printing. If the control character is a +, the carriage will not advance. (See figure 4-8.)

It is recommended that a carriage control character always be included in a WRITE FORMAT statement. This ensures that the line printer is always under programmer control, and prevents inadvertent line skipping and the like.

The use of carriage control characters is illustrated in figure 4-9. The use of a literal to establish the carriage control character is an acceptable alternative to the example shown in figure 4-9. Figure 4-10 shows the use of a literal as the carraige control character.

4.6 Multiple Record Format Statement

In previous instruction it was implied that the right parenthesis indicated the end of the format specification and hence the end of the data being read,

Character	Vertical Spacing Before Printing
Blank	One Line
0	Two Lines
1	To First Line of Next Page
+	No Advance

Figure 4-8.–Carriage control characters.

Statements	Explanation
`1 FORMAT (1H1, 10X, I2)` `WRITE (6, 1) N`	The printer will skip to the top of a new page, skip 10 spaces on the first line of that page and write the value of N in the next two spaces.

Figure 4-9.–Use of carriage control characters.

```
10 FORMAT('0',I4,5X,I3)

20 FORMAT(' 1979 MANPOWER REPORT')
```

The first example shows the carriage control character for double spacing, while the second example includes the blank as part of the literal to be used as a report heading. Remember, the first character will be used as the carriage control character.

Figure 4-10.—Use of a literal for carriage control.

or the end of the printed line. If items of the read or print list remained to be considered at that point, the format was repeated from the left parenthesis using the next data card or print line.

In FORTRAN the slash (/) may also be used to indicate the end of a record. The statements

```
33 FORMAT (F20.8,2F12.4,/,5F20.8)

READ (5,33) A,B,C,V,W,X,Y,Z
```

would read one data card with the format F20.8, F12.4, F12.4; then, would read the next card with the format F20.8, F20.8, F20.8, F20.8, F20.8. Thus, data cards having different data layouts can be read with ease.

When slashes are used in the middle of a WRITE FORMAT statement then the number of print lines (or records) skipped equals the number of slashes minus one. Therefore, the statements

```
WRITE (6,10)J,K

10 FORMAT (1H0,I3,///,I5)
```

would cause the value of J to be printed on the first line, then the printer would skip 2 lines, and the value of K would be printed on the 4th line, beginning in column 1.

When slashes are used at the beginning or end of a WRITE FORMAT statement, the number of print lines skipped equals the number of slashes used. Therefore, the statements

```
WRITE (6,12)J

12 FORMAT (' ',I3,///)
```

would cause the value of J to be printed on the first line, then the printer would skip three lines.

CHAPTER 4

EXERCISES

1. Given: A keypunched data card containing the following figures.

Columns	1	2	3	4	5	6	7	8	9	10
Data	7	9	8	6	1	3	4	2	0	5

Required: Identify the values that would be assigned to the variables indicated by each of the following FORMAT – READ combinations.

a. 10 FORMAT (I4)

READ (5,10)K ______________________

b. 11 FORMAT (2I4)

READ (5,11)K, L ______________________

c. 12 FORMAT (3I3)

READ (5,12)K, L, M ______________________

d. 13 FORMAT (F5.2)

READ (5,13)A ______________________

e. 14 FORMAT (2F5.3)

READ (5,14)A,B ______________________

f. 15 FORMAT (3F3.2)

READ (5,15) A,B,C ______________________

2. Write F specifications for the following numbers.

a. 234.78 ______________________

b. 4218.9 ______________________

c. .00049 ______________________

d. 3875.34 ______________________

e. 856. ______________________

f. 1.0 ______________________

CHAPTER 4

EXERCISES–Continued

3. Write the H specification for the following:

   a. THE TOTAL NUMBER IS

   b. NAMEbbbbbbbbbRANKbbbbbbSERVICE NUMBER

4. Given:

First Data Card

Columns	1	2	3	4	5	6	7	8	9	10	11
Data	b	b	-	3	2	1	4	5	2	b	b

(Note: The b symbol is a common way of showing blank spaces.)

Second Data Card

Columns	1	2	3	4	5	6	7	8	9	10	11
Data	b	b	b	b	3	2	4	9	1	2	b

Write the coding needed to read the value on the first data card, as ITEM (integer); then, read the value on the next card as LIST (real with two decimal places). Then, print the two values on the same line.

> Throughout the course exercise questions such as questions 4 and 5 of Chapter 4 require the coding of FORTRAN statements to answer the question properly. Blank coding sheets will follow the question. Use these blank coding sheets to develop your answer.

CHAPTER 4

EXERCISES–Continued

COMM.	STATEMENT NUMBER				CONT.	FORTRAN STATEMENT																																													
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52

CHAPTER 4

EXERCISES–Continued

5. Given:

Data Card

Columns	1	2	3	4	5	6	7	8	9	10	11	12	13
Data	-	4	3	.	4	b	b	b	+	6	7	2	b

Write the coding needed to read the two values on this data card as the real variable BIG and the integer variable WHEEL. Then, print BIG in the 42nd column of a print line, skip three print lines and print WHEEL in the 16th column of a print line.

COMM.	STATEMENT NUMBER	CONT.	FORTRAN STATEMENT
1	2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52

CHAPTER 4

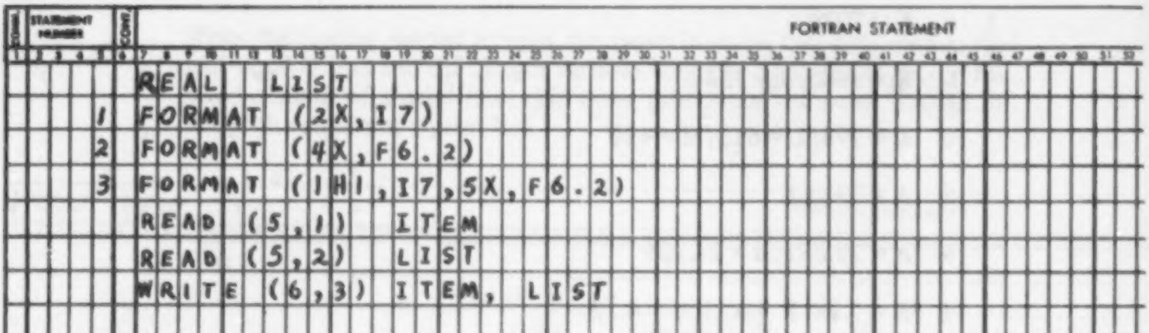
EXERCISE SOLUTIONS

1. a. K = 7986
   b. K = 7986; L = 1342
   c. K = 798; L = 613; M = 420
   d. A = 798.61
   e. A = 79.861; B = 34.205
   f. A = 7.98; B = 6.13; C = 4.20
2. a. F6.2
   b. F6.1
   c. F6.5
   d. F7.2
   e. F4.0
   f. F3.1
3. a. 19HTHE TOTAL NUMBER IS
   b. 4HNAME,10X,4HRANK,6X,14HSERVICE NUMBER

CHAPTER 4

EXERCISE SOLUTIONS–Continued

4.

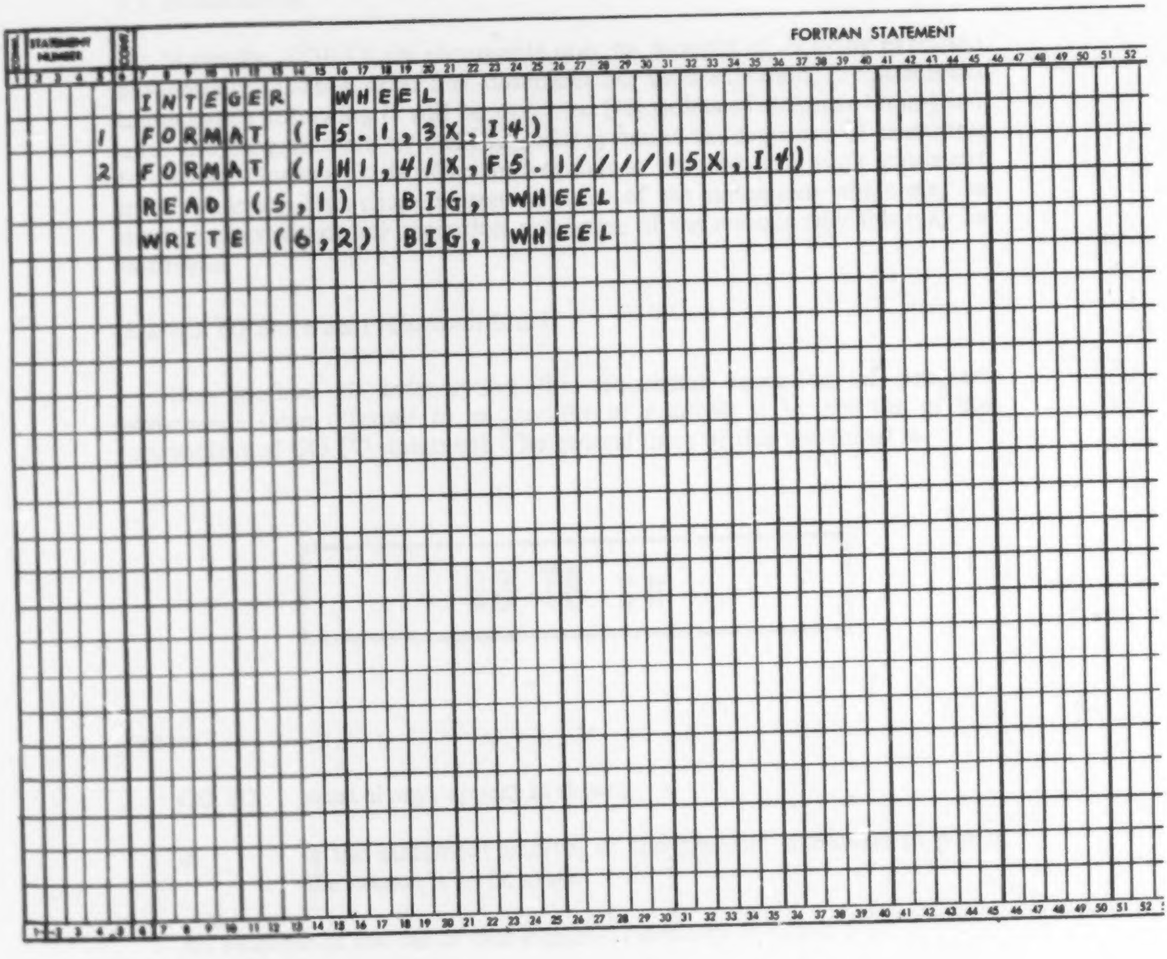

STATEMENT NUMBER
FORTRAN STATEMENT

```
      REAL    LIST
    1 FORMAT  (2X,I7)
    2 FORMAT  (4X,F6.2)
    3 FORMAT  (1H1,I7,5X,F6.2)
      READ  (5,1)   ITEM
      READ  (5,2)   LIST
      WRITE  (6,3)  ITEM,  LIST
```

CHAPTER 4

EXERCISE SOLUTIONS–Continued

5.



FORTRAN STATEMENT

```
      INTEGER  WHEEL
    1 FORMAT (F5.1,3X.I4)
    2 FORMAT (1H1,41X,F5.1////15X,I4)
      READ (5,1)  BIG, WHEEL
      WRITE (6,2) BIG, WHEEL
```

CHAPTER 5

CONTROL STATEMENTS

5.1 Introduction

Normally, FORTRAN statements may be thought of as being executed sequentially. That is, after one statement has been executed, the statement immediately following it will be executed (regardless of statement numbers) until the program ends or until something causes the sequence of execution to change. However, it is often undesirable to proceed with each statement in this manner. This chapter presents some of the statements which may be used to alter sequential execution and some of the reasons why this may be desirable.

5.2 GO TO Statement (Unconditional)

One method of interrupting the sequential execution of program statements (also referred to as transfer of control) is by the use of the unconditional GO TO statement. The general form of this statement is–

```
GO TO sn
```

where

GO TO	must always appear as shown.
sn	is the statement number of the program statement to which the branch is to be made.

An example of the use of this statement is shown in figure 5-1.

5.3 The Arithmetic IF Statement

Perhaps the most powerful tool in the FORTRAN repertoire of instructions is the IF statement, for it is this statement that allows the

Statements	Explanation

`xxxxxxxxxxxxxxx`
`xxxxxxxxxxxxxxx`
`GO TO 9`
`xxxxxxxxxxxxxxx`
`9 xxxxxxxxxxxxxxx`
	This statement causes the program to branch to the procedure defined by statement number 9.

Figure 5-1.—Use of the GO TO statement.

computer to make decisions. Two types of IF statements are examined in this manual: the arithmetic IF statement and the relational IF statement.

The general form of the arithmetic IF statement is–

$$\boxed{\text{IF }(e)\ sn_1\ ,\ sn_2\ ,\ sn_3}$$

where

IF	must appear as shown.
(e)	is any FORTRAN expression.
sn_1	is the statement number of the program statement to which control is transferred if the value of the expression (e) is less than zero.
sn_2	is the statement number of the program statement to which control is transferred if the value of the expression (e) is zero.
sn_3	is the statement number of the program statement to which control is transferred if the value of the expression (e) is greater than zero.

An example of the use of this statement is shown in figure 5-2.

Statement	Explanation
`IF (X-10.5) 10, 5, 15`	This statement transfers control to statement 10 if the value of X-10.5 is less than zero; it transfers control to statement 5 if the value of X-10.5 is zero; or it transfers control to statement 15 if the value of X-10.5 is greater than zero.

Figure 5-2.–Use of the arithmetic IF statement.

5.4 The Relational IF Statement

The general form of the relational IF statement is–

```
IF (t)   S
```

where

IF	must appear as shown.
(t)	is any relational expression using one of the relational operators given in figure 5-3 (A.GT.B; A.EQ.B; etc.).
S	is the next program statement to be executed if the expression (t) is true. S can be any statement other than another relational IF statement or a DO statement (to be covered in Section 5.6). If (t) is not true, the next sequential statement in the program is executed.

An example of the use of this statement is shown in figure 5-4.

.GT.	Greater Than
.GE.	Greater Than or Equal To
.LT.	Less Than
.LE.	Less Than or Equal To
.EQ.	Equal To
.NE.	Not Equal To

(Note that the operators must always be preceded and followed by a period.)

Figure 5-3.—FORTRAN relational operators.

Statement	Explanation
IF (A.GT.B) GO TO 13 STOP 13 WRITE (6, 2) ALPHA	This IF statement directs that the next statement to be executed is GO TO 13, if A is greater than (.GT.) B. If A is equal to or smaller than B, then the procedure defined by the statement following the IF statement (STOP in this instance) is executed.
* * * * * *	
IF (A.GT.B) TOTAL = A GO TO 10	This IF statement directs that the next statement to be executed is TOTAL = A, if A is greater than B. Then the statement GO TO 10 is executed. If A is equal to or smaller than B, then the procedure defined by the statement following the IF statement (GO TO 10 in this instance) is executed.

Figure 5-4.—Use of the relational IF statement.

FORTRAN also allows for more complex relational expressions through the use of two additional operators: .AND. and .OR. The use of these operators is illustrated in figure 5-5.

5.5 Computed GO TO Statements

It is often advantageous to transfer control to different parts of the program depending on the value of some variable. This could be accomplished through a series of IF statements; however, with the use of the

Statement	Explanation
IF (A. EQ. B. AND. X. EQ. Y) AMT = B*. 50*Y GO TO 5	This IF statement directs that the next statement to be executed is AMT=B*. 50*Y, if both relational expressions are true. If either or both of the relational expressions are not true, then the procedure defined by the statement following the IF statement (GO TO 5 in this instance) is to be executed.
* * * * * *	
IF (A. EQ. B. OR. X. EQ. Y) GO TO 13 GO TO 5	This IF statement directs that the next statement to be executed is GO TO 13 if either or both relational expressions are true. If neither of the relational expressions are true, then the statement following the IF statement is to be executed.

Figure 5-5.–The complex relational IF statement.

computed GO TO, one statement can replace a whole series of IF statements. This conditional form of the GO TO statement should be contrasted with the unconditional GO TO in Section 5.2.

The general form of the computed GO TO is–

$$\boxed{\text{GO TO } (sn_1\ ,\ sn_2\ ,\ \ldots,\ sn_m),\ i}$$

where

GO TO	must appear as shown.
$(sn_1, sn_2, \ldots sn_m)$,	represents program statement numbers. The parentheses and the commas are required as shown.
i	represents a nonsubscripted integer variable that is greater than zero, but not greater than m.

Transfer of control is made to sn_1 if i=1, to sn_2 if i=2, to sn_m if i=m, and so forth. The use of this statement is shown in figure 5-6.

If i does exceed m, transfer of control simply continues with the next instruction following the computed GO TO statement.

Statement	Explanation
GO TO (10, 5, 15), N	This statement causes a transfer of control to statement 10 if N=1 (because the number 10 is in position 1 inside the parentheses), to 5 if N=2, or to 15 if N=3. In this example the value of N should not be greater than 3.

Figure 5-6.–The computed GO TO statement.

5.6 DO Statement

Another powerful tool available in FORTRAN is the DO statement, which is a command to execute repeatedly a series of procedures. It is very useful for building a loop in a program, frequently called a "DO loop."

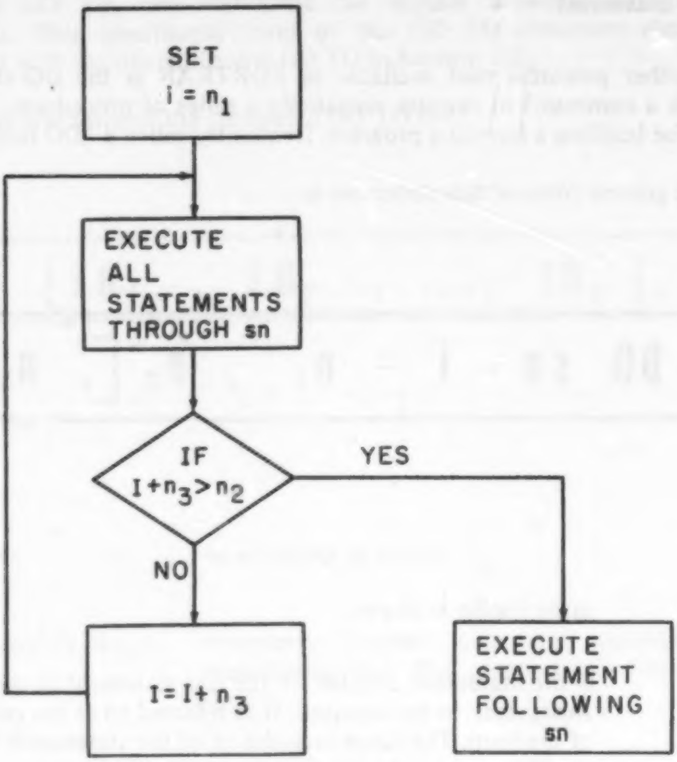
The general form of this statement is–

$$\boxed{\mathrm{DO}\ \ sn\ \ \ i\ =\ n_1\ \ ,\ n_2\ [,\ n_3]}$$

where

DO	must appear as shown.
sn	is the statement number of the last statement in the series of statements to be repeated; it is referred to as the range limits of the loop. The range includes all of the statements following the DO, up to and including sn.
i	is an integer variable whose value is controlled by n_1, n_2, and n_3.
n_1	is an unsigned integer constant or variable; it specifies the initial value of i.
n_2	is an unsigned integer constant or variable; it specifies the final value of i.
n_3	is an unsigned integer constant or variable; it specifies the increment by which i is increased each time the series is repeated. If the increment value is one, n_3 may be omitted leaving the statement in the form DO sn i = n_1, n_2.

On the first execution of the DO loop i is set equal to n_1. On each repetition thereafter, i is increased by the increment n_3. After the loop is executed with i equal to n_2, the procedure defined by the statement following the last statement in the series is executed. This is called the normal exit from the loop. A flowchart of this procedure is shown in figure 5-7.



NOTE: If n_3 is not used, the value 1 is substituted for n_3.

Figure 5-7.—Logical flow of the DO statement.

An example of the DO statement is shown in figure 5-8.

The following rules apply to the use of DO loops and must be carefully observed:

RULE 1: The first statement within a DO loop must be an executable statement. For example, a FORMAT statement is nonexecutable and could not be the first statement because it only provides information to the compiler. Contrast this to a statement such as a READ or WRITE which causes the compiler to produce machine language instructions to accomplish the specified procedure.

RULE 2: No statement within a loop can change the values of the i, n_1, n_2, or n_3 parameters. However, there is no restriction on their use within the loop as long as their values remain the same. (See figure 5-9.)

Statement

```
      DO 30 K = 1, 10, 1
      READ (8, 1) X
      Y = X + 2
      WRITE (6, 2) Y
   30 CONTINUE
```

Explanation

This statement causes the execution of all statements from the DO statement through the CONTINUE statement ten times.

For the first execution, K=1; for the second execution, K=2, and so forth. When K=10, the loop is performed the last time. Control is then passed to the next statement after CONTINUE.

Figure 5-8.—The DO statement.

Statement

```
      DO 5 J = 1, 8
      SUM = M**J
    5 CONTINUE
```

Explanation

This statement causes the execution of all statements from the DO statement through the CONTINUE statement, 8 times. For the first execution, J=1. When J=8, the loop is performed the last time.

J is being used here as an exponent with its current value as supplied by the DO statement. That is, M will be raised to the first power during the first execution, to the second power during the second execution, and so forth.

Note that the value of J is not affected in any way by its use in the statement SUM = M**J.

Figure 5-9.—The use of index parameters as loop variables.

The current values of i, n_1, n_2, and n_3, are also available for use outside the DO loop if an exit other than normal is made. Recall that a normal exit refers to the transferring of control out of the DO loop after it has been executed n_2 times. This feature can be an extremely valuable tool to the programmer. An other than normal exit can be accomplished by a GO TO statement or arithmetic IF statement. The current values of i, n_1, n_2, and n_3 could then be used as any variable.

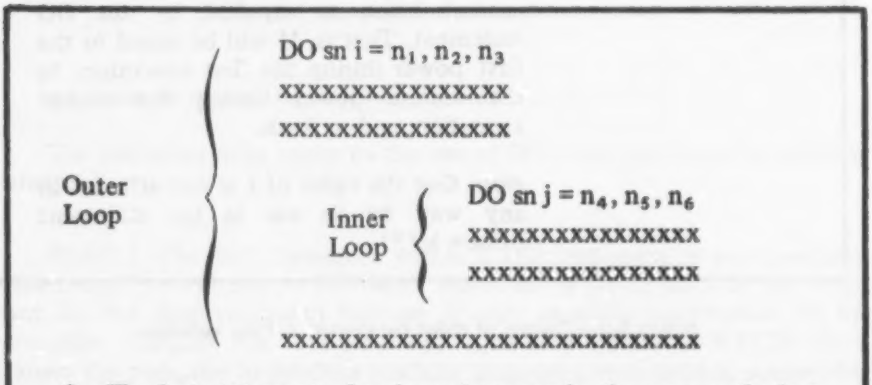
RULE 3: FORTRAN permits the nesting of DO loops (or loops within loops). However, all of the statements of an inner loop must be within the range of an outer loop. (See figure 5-10.)

RULE 4: It is permissable to transfer control from an inner loop to an outer loop, but it is not permitted to transfer control into the range of an inner DO loop.

RULE 5: No DO loop can end on a change of control statement. When a DO loop reaches the last statement in its range, control is transferred back to the first statement and the loop cycles in this manner until it executes the prescribed number of cycles or until control is transferred completely outside of the loop. If the last statement of the loop were a change of control, such as a GO TO or an IF, program control would have to be transferred not only back to the start of the DO loop, but also to the statement specified by the control statement. Such a condition would create an error during compilation.

5.7 CONTINUE Statement

In order to avoid the confusion over writing an acceptable end statement for a DO loop, the statement CONTINUE has been provided. CONTINUE is a dummy statement that has no effect on the program other than to provide


DO sn i = n_1, n_2, n_3
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
Outer Loop
Inner Loop
DO sn j = n_4, n_5, n_6
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

* The last statement of an inner loop can be the same as the last statement of an outer loop. n_4, n_5, and n_6 serve the same purpose as n_1, n_2, and n_3.

Figure 5-10.—Nested DO loops.

a good end statement for DO loops. The general form of the DO loop then becomes–

$$\text{DO}\ \ sn\ \ i = n_1\ ,\ n_2 \left[,\ n_3\right]$$
$$sn\ \ \text{CONTINUE}$$

It is a good practice always to define explicitly the end of a DO loop with a CONTINUE statement.

The use of the CONTINUE statement is shown in figure 5-11.

5.8 STOP Statement

This statement stops the execution of the object program at the point at which it appears. Either a STOP statement or an equivalent statement (such as a RETURN statement, which is explained below) must appear at the logical termination of each FORTRAN source program. (See figure 5-12.)

Statement	Explanation
DO 30 K = 1, 10, 1 READ (8, 1) X A = X**2 WRITE (6, 2) A 30 CONTINUE	This program segment will cause the computer to read a value of X, calculate A, and write A for 10 values of X. It will then execute subsequent program procedures.

Figure 5-11.–Use of the CONTINUE statement.

With STOP Statement	With RETURN Statement
xxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxx STOP END	xxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxx RETURN END

Figure 5-12.–Source program ending.

5.9 RETURN Statement

When used at the logical termination of a main program, this statement serves the same purpose as a STOP statement. (See figure 5-12.)

However, unlike the STOP statement, it does not stop the computer. Thus, it is also used in subprograms in order to transfer control back to the main program to the point from which the subprogram was called.

5.10 END Statement

This statement identifies the last statement of every source program to the compiler. It must be written as the last statement of the source program and can be located anywhere between columns 7 and 72 on the coding sheet. (See figure 5-12.)

CHAPTER 5
EXERCISES

1. Fill out the IF statement so control goes to 20 if e is negative, to 10 if e is zero, and to 30 if e is positive.

```
10 *********
   *********
   *********
   IF (e) ______________, ______________, ______________
   *********

20 *********
   *********

30 *********
   *********
```

2. Write a program segment that will add 1 to M1 if KODE = 1, will add 1 to M2 if KODE = 2, or will add 1 to M3 if KODE = 3. Then, transfer control to statement number 5 in each case.

STATEMENT NUMBER

CONT.

FORTRAN STATEMENT

CHAPTER 5

EXERCISES–Continued

3. What is the value of K after each of the following DO loops have been executed?

a.
```
 K = 10
    DO 16 I = 1, 10, 5
16 K = K + 2
```

b.
```
 K = 10
    DO 16 I = 1, 7, 2
16 K = K * I
```

c.
```
 K = 40
    DO 16 I = 2, 6, 2
    N = I/2
16 K = N * K
```

4. If I has been set to 30, what is the value of I after the following sequence has been executed?

```
    IF (NUMBER -5) 10, 20, 30
10 I = I + 10
    GO TO 5
20 I = I + 16
    GO TO 5
30 I = I * 5
 5
```

FOR NUMBER =

a. -11 ____________________

b. 5 ____________________

c. 20 ____________________

d. -5 ____________________

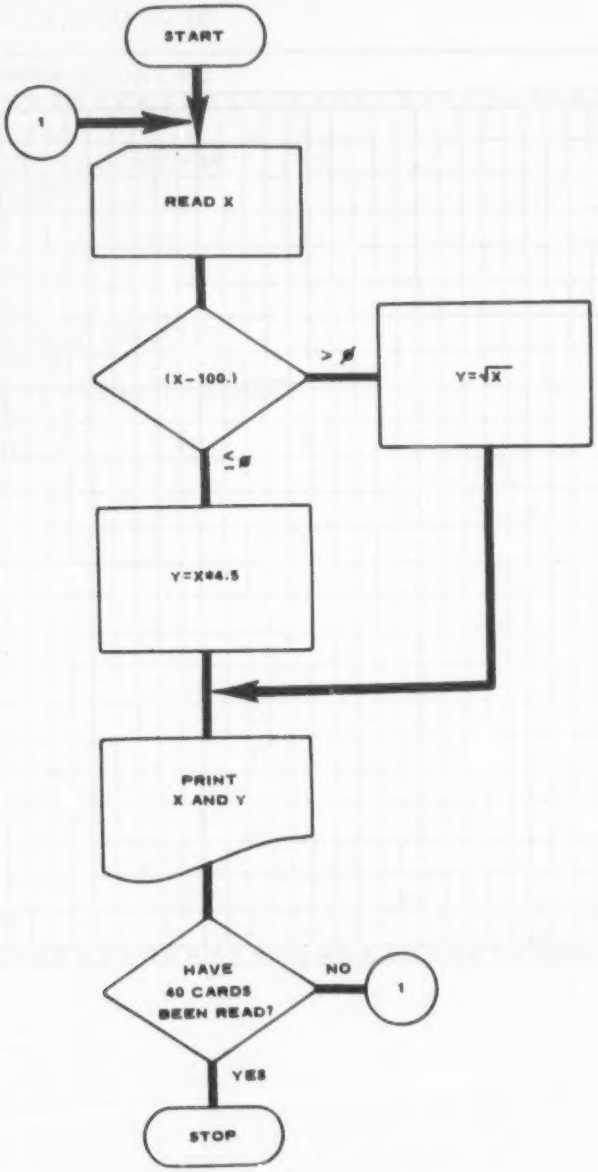
e. 6 ____________________

f. 0 ____________________

CHAPTER 5

EXERCISES–Continued

5. Using the flowchart provided, write a program that will read in values of X from 40 cards. When X is greater than 100, $Y = \sqrt{X}$. When X is equal to or less than 100, Y = X * 4.5. X is in columns 1 through 4 of each card read, is signed, and has no fractional part. A decimal is not punched in the card. Print each value of Y with its corresponding X. Allow for a Y of ± 9999.99 and print X with a decimal point.



CHAPTER 5

EXERCISES–Continued

COMMENT	STATEMENT NUMBER	CONT.	FORTRAN STATEMENT
1	2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
1	2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52

CHAPTER 5

EXERCISES–Continued

6. What is in KOUNT after each of the following programs is executed?

a.

```
    KOUNT = 0
    M = 2
    N = M + 2
    IF (N.LE.5) GO TO 10
    M = N
    KOUNT = KOUNT + 2
    GO TO 20
 10 M = 2 * N
    KOUNT = KOUNT + M
 20 RETURN
    END
```

b.

```
    KOUNT = 2
    M = 5
 10 IF (M.EQ.3) GO TO 20
    M = M - 1
    KOUNT = KOUNT * KOUNT
    GO TO 10
 20 RETURN
    END
```

CHAPTER 5

EXERCISE SOLUTIONS

1. 20, 10, 30

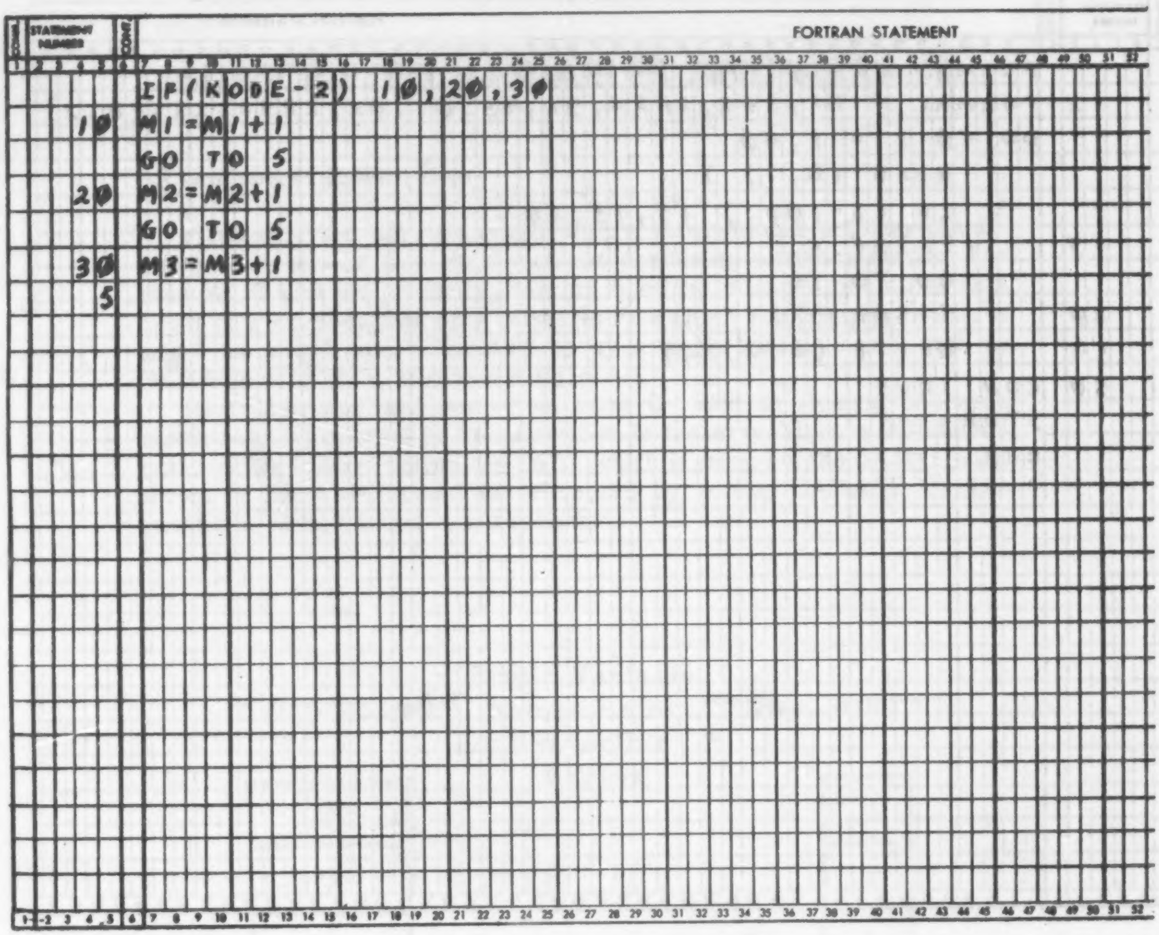
2. Solution 1: Using the computed GO TO statement

```
      GO TO (10,20,30), KODE
   10 M1=M1+1
      GO TO 5
   20 M2=M2+1
      GO TO 5
   30 M3=M3+1
    5
```

CHAPTER 5

EXERCISE SOLUTIONS–Continued

Solution 2: Using the arithmetic IF statement


STATEMENT NUMBER
CONT.
FORTRAN STATEMENT

```
      IF(KODE-2) 10,20,30
   10 M1=M1+1
      GO TO 5
   20 M2=M2+1
      GO TO 5
   30 M3=M3+1
    5
```

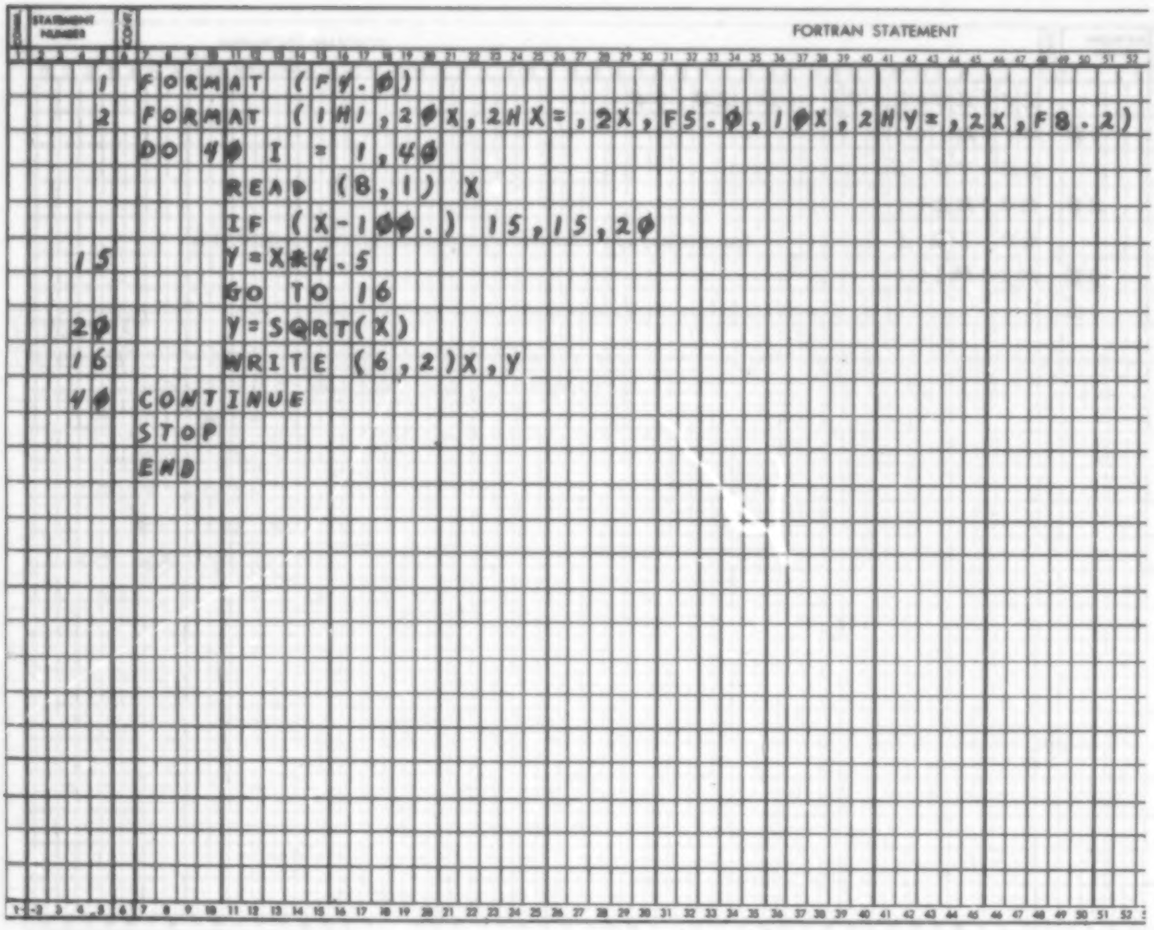
3. a. K = 14
   b. K = 1050
   c. K = 24

4. a. 40
   b. 46
   c. 150
   d. 40
   e. 150
   f. 40

CHAPTER 5

EXERCISE SOLUTIONS–Continued

5.


STATEMENT NUMBER
FORTRAN STATEMENT

```
    1 FORMAT (F4.0)
    2 FORMAT (1H1,20X,2HX=,2X,F5.0,10X,2HY=,2X,F8.2)
      DO 40 I = 1,40
          READ (8,1) X
          IF (X-100.) 15,15,20
   15     Y=X*4.5
          GO TO 16
   20     Y=SQRT(X)
   16     WRITE (6,2)X,Y
   40 CONTINUE
      STOP
      END
```

6. a. KOUNT = 8
   b. KOUNT = 16

CHAPTER 6

SUBSCRIPTED VARIABLES AND ARRAYS

6.1 Grouping of Related Data

Frequently the data associated with a programming problem can be arranged in sets of data with each set containing related elements. For example, if you are processing data on bombing missions, one set of data might be a list of damage percentages. In FORTRAN this grouping of data is called an array* and is referred to by a single variable name, such as DAMAGE or SORTY, as shown in figure 6-1.

Each individual member of an array is called an element and each element can be referred to by its position in the array. Thus, in FORTRAN notation the damage percentage "67," which is second in the DAMAGE list shown in figure 6-1, could be referenced by writing DAMAGE (2). The quantity–(2)–is referred to as a subscript.

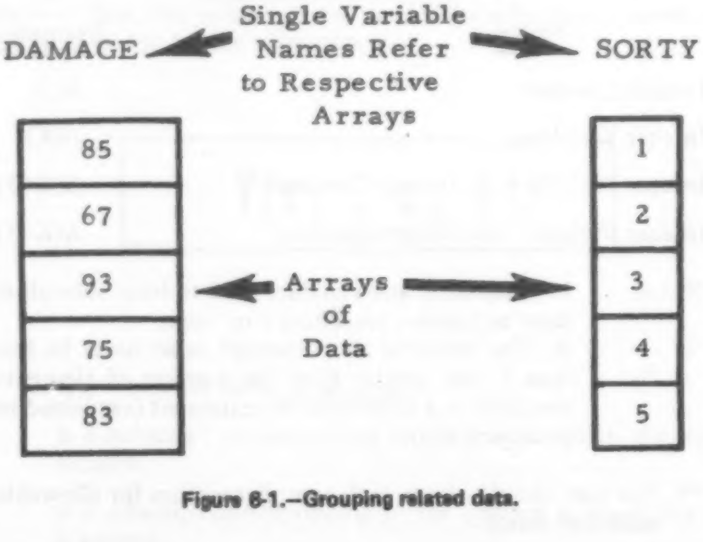


Figure 6-1.–Grouping related data.

---

*Note: In COBOL such groupings of related information are called tables.

6.2 Subscripts

Much of the power of FORTRAN is found in its ability to handle variables with one or more subscripts. This ability enables the programmer to handle matrix operations efficiently, which is an important feature for a language oriented toward mathematical and scientific applications.

The general form of a single subscript is–

```
V(s)
```

where

V	represents the name of an array.
s	is an allowable form of a FORTRAN subscript; it gives the position of an element in the array.

Some of the allowable forms of a subscript are shown in figure 6-2.

A variable subscript must always be set to some positive value prior to its use in the program. The use of the FORTRAN subscript is illustrated in figure 6-3.

Allowable Subscript Forms**

Forms	Examples
An Integer Constant	A(2)
An Integer Variable	A(K)
An Integer Variable + An Integer Constant	A(K+3)
An Integer Variable - An Integer Constant	A(K -3)

Notes:
1. Constants and variables used to form subscripts must not have a preceding + or - sign.
2. The value of the subscript must never be less than 1, nor greater than the number of elements specified in a DIMENSION statement (explained in paragraph 6.6).

** The user should check each compiler version for allowable subscript forms.

Figure 6-2.–Allowable subscript forms**.

Problem: Find the sum of 1000 numbers which are stored in the computer in an array named A.

Solution

```
 I = 1
 NUMSUM = 0
1 NUMSUM = NUMSUM + A (I)
 IF (I. EQ. 1000) GO TO 2
 I = I + 1
 GO TO 1
2 STOP
```

Explanation

The programmer could add A(1) + A(2) + A(3) . . . A (1000) to achieve the result.

However, it is much more convenient to use a variable subscript A(I) and arrange for I to take on all values from 1 to 1000. Note that the subscript I is set to its initial value prior to its use in the program.

Figure 6-3.–Use of subscripts.

6.3 Double Subscripts

When more than one list of information is involved in an array, as shown in figure 6-4, then two subscripts are needed to refer to any single element. The general form for double subscripts is–

$$\boxed{V(s_1 , s_2)}$$

where

V represents the name of an array.

s_1 is a subscript corresponding to the row in which the element is located.

s_2 is a subscript corresponding to the column in which the element is located.

The comma and parentheses are required as shown.

	Column 1	Column 2	Column 3
Row-1	10	105	99
Row-2	15	75	4
Row-3	6	63	52

Figure 6-4.—Two-dimensional arrays.

The use of double subscripts is shown in figure 6-5.

6.4 Three-Dimensional Arrays

Most FORTRAN compilers also permit the use of three-dimensional arrays, and some permit even more complicated arrays. However, this feature is beyond the scope of this manual.

Examples	Explanation
A (2, 4)	Refers to the element of array A, which is located in the second row and fourth column.
NUMBER (1, 8)	Refers to the element of array NUMBER, which is located in the first row and eighth column.

Figure 6-5.—Use of double subscripts.

6.5 DIMENSION Statement

Any subscripted variable appearing in a program must be previously mentioned in a DIMENSION statement. This statement is used to tell the computer how many memory locations are required to store a given array.

The general form of this statement is–

DIMENSION VAR1(s_1 ,....s_n), VAR2(s_1 ,....s_n),.....

where

DIMENSION	must appear as shown.
VAR1, VAR2, etc.	are the names of the arrays.
$s_1, \ldots\ldots\ldots s_n$	are the highest values of the corresponding subscripts and must be integer constants.

A comma is required between each set of array information, as shown in the preceding general form.

The actual number of storage locations reserved for each array can be determined by the following formula, $s_1 \text{ X } s_2 \ldots \text{ X } s_n$. Therefore, an array–A(10,5)–would require 10 X 5 or 50 storage locations.

The use of the DIMENSION statement is shown in figure 6-6.

Statements	Explanations
DIMENSION A(10)	The A array has ten elements and requires that many memory locations.
DIMENSION LIST (10), GROUP (5, 5), TABLE (5, 4, 3)	This statement would set aside 10 memory locations for the subscripted variable LIST, 25 locations for GROUP, and 60 for TABLE.

Figure 6-6.–Use of the DIMENSION statement.

Arrays can store either fixed- or floating-point data. The rules for specifying array names are the same as for other variable names.

If a subscripted variable appears in a program as, for example, COST (J + 1), the DIMENSION statement cannot be written as DIMENSION COST (J + 1). Instead, the highest value of J + 1 must be determined and used in the DIMENSION statement.

The DIMENSION statement is considered a nonexecutable statement just as the FORMAT statement is. In addition, the DIMENSION statement need not be assigned a statement number. Since the DIMENSION statement is a specification statement it must appear in a program according to the rules in Section 1.6.

6.6 Storing of an Array

The FORTRAN user should be aware of this aspect of arrays, as it affects the way in which data is written into and extracted from arrays. The elements of an array are stored in sequential memory locations of increasing address. The programmer can explicitly control the sequence by which elements of an array are stored (or printed) by using the subscripts under the control of DO loops (as discussed in Section 6.8). However, it is also possible to read or print arrays without specifying subscripts; in this case the computer stores the elements in the following manner.

As far as one-dimensional arrays are concerned, each element of the array is stored in order of appearance, as shown in figure 6-7. If data is

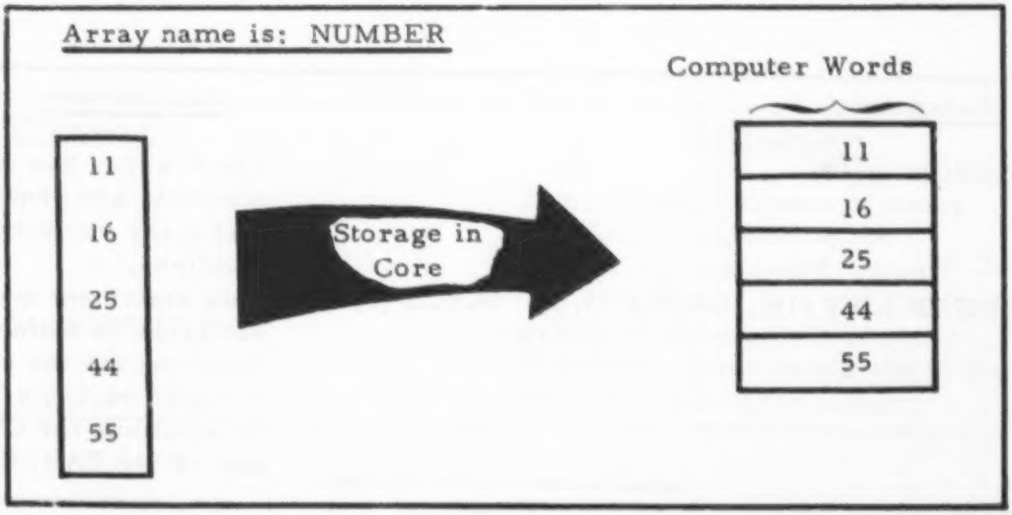


Figure 6-7.—Computer-controlled storage of one-dimensional arrays.

being read from cards, then each element is stored in the array in accordance with its position on the card. If data is being printed from memory, then it appears on the printed page in the same sequence in which it was stored (element-1, element-2, . . . element-n).

Two-dimensional arrays are treated in much the same way as one-dimensional arrays, in that each column of the array is treated as a single array. This means that the elements are stored (or extracted) by column. Therefore, for the array in figure 6-8, each element is stored in the following order:

N(1,1)
N(2,1)
N(3,1)
N(1,2)
N(2,2)
N(3,2)
N(1,3)
N(2,3)
N(3,3)

Note that the first subscript (which corresponds to the row) advances more rapidly than the second subscript (which corresponds to the column).

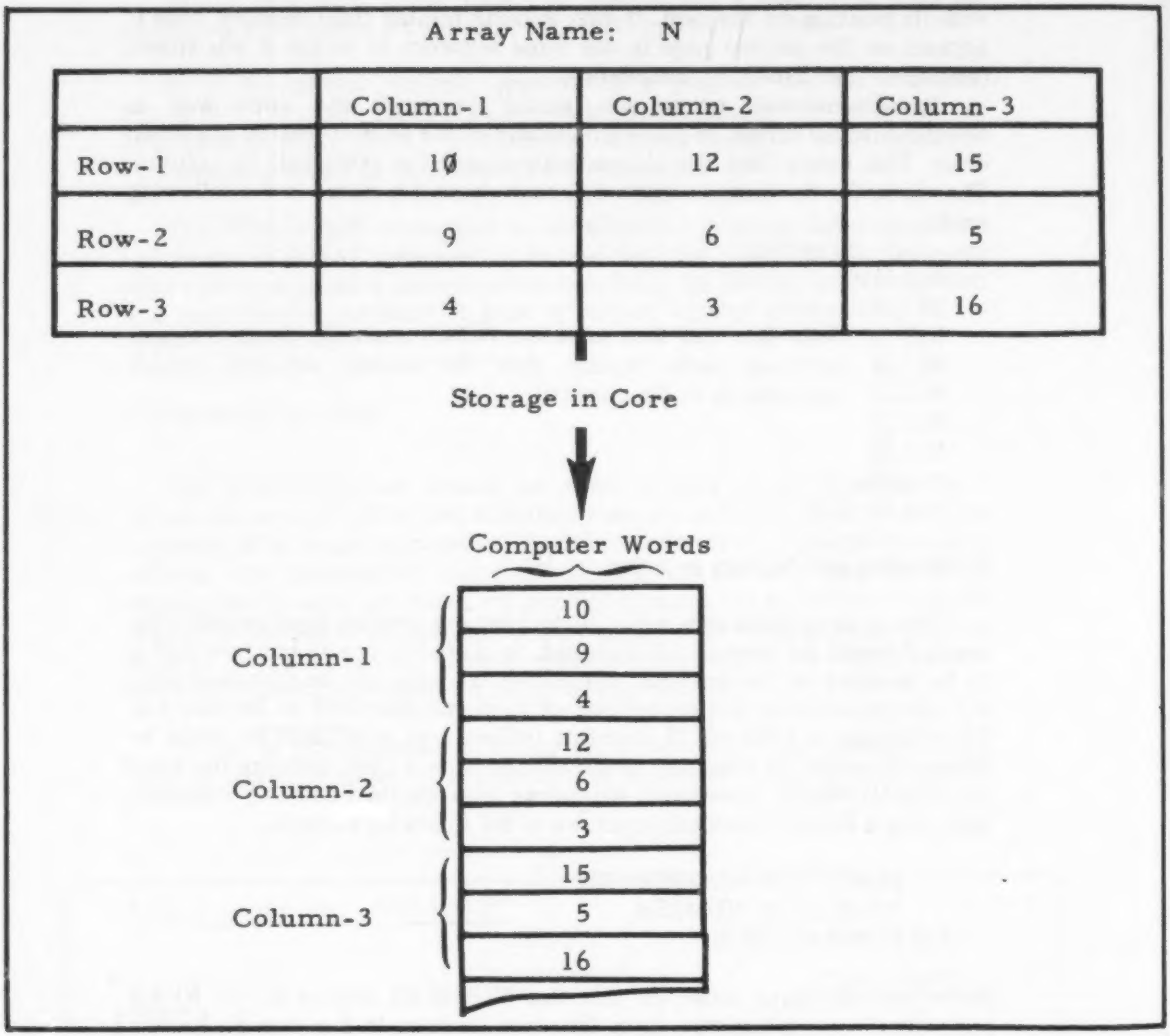
6.7 Reading and Printing an Array

You must exercise care when reading into or printing from an array, for much depends on how data is punched on cards (for input) and how data is to be arranged on the printout. Sometimes the array can be processed using the computer-controlled sequencing of elements described in Section 6.6. For example, a table of 20 elements (referred to as NUMBER) could be filled efficiently by punching all the elements on a card, defining the array by a DIMENSION statement, writing an appropriate FORMAT statment, and using a READ statement, as shown in the following example.

```
   DIMENSION NUMBER (20)
   READ (5,10) NUMBER
10 FORMAT (20I2)
```

Note that the array name (in this case NUMBER) appears in the READ statement in nonsubscripted form. However, its usage in this example has the same effect as writing NUMBER (1), NUMBER (2), ... NUMBER (20), as each quantity is stored in that order. Similarly, it might be desirable in some instances to print an entire array all at one time in a report, and again one could effectively rely on computer-controlled sequencing in order to extract each element.

However, normally the programmer has to control the processing of array information explicitly. For example, quantities might appear on a punched card in a different order than is desired; arrays might not be filled all at once and thus elements might be read into storage at different points in the program; or reports might be produced using selected elements only from particular arrays. When faced with such cases as these, the programmer must use array names in subscripted form. Some of the ways that such a subscripted array name might appear are examined in the following paragraphs.



Array Name: N

	Column-1	Column-2	Column-3
Row-1	10	12	15
Row-2	9	6	5
Row-3	4	3	16

Figure 6-8.–Computer-controlled storing of two-dimensional array.

Suppose a one-dimensional array (A) of 10 numbers is to be read. If there is one value per card, then the statements

```
10 FORMAT (F6.3)
   DO 3 I = 1, 10
   READ (5, 10) A(I)
 3 CONTINUE
```

would read the values from 10 cards and assign them as A(1), A(2), A(3), and so forth.

In order to read the array when all 10 values are placed on one card, the following statements could be used–

```
10 FORMAT (10F6.3)
   READ (5, 10) A(1), A(2), A(3), . . . A(10)
       or
10 FORMAT (10F6.3)
   READ (5, 10) A
```

Note that the first set of statements would be very awkward if you were working with a very large array, say 100 or 1,000 elements. Therefore, another form is used which is much more convenient but which still provides the programmer with explicit control of the ordering of elements.

This form is referred to as the IMPLICIT DO loop and is illustrated by the following solution to the preceding problem.

```
10 FORMAT (10F6.3)
   READ (5, 10) (A(I), I = 1, 10)
```

When written as shown, "I" assumes all integer values from 1 to 10. Notice that the subscripted variable, A(I), and the IMPLICIT DO loop, I = 1, 10, are enclosed in parentheses. All subscripted variables within the parentheses are affected by the DO loop. The IMPLICIT DO loop is an extremely valuable tool in FORTRAN and finds wide application in both input and output processes.

The general form of the IMPLICIT DO loop for single-subscripted variables is–

$$\boxed{(V(s_1\ ),\ s_1 = n_1,\ n_2,\ n_3)}$$

where

V	is the variable name of the array.
s_1	represents the subscript.
n_1	is the initial value for the subscript.
n_2	is the final value for the subscript.
n_3	is the increment value; as with a normal DO loop, n_3 can be omitted if the increment is 1.

All punctuation must be included as shown.

The general form of the IMPLICIT DO loop for double-subscripted variables is–

$$\boxed{(V(s_1\ ,\ s_2),\ s_i = n_1\ ,\ n_2\ ,\ n_3)}$$

where

V, s_1, n_1, n_2, n_3	have the same meaning as used for single-subscripted variables.
s_2	represents the second subscript.
s_i	is the subscript (s_1 or s_2) being varied.

Again, all punctuation must be included as shown.

Consider the case where a two-dimensional array, A(I,J), is to be read. If there were one value per card, either of the following would suffice:

```
10 FORMAT (F6.3)
   DO 1 I = 1, 10
   DO 1 J = 1, 10
   READ (5, 10) A(I,J)
 1 CONTINUE
```

or

```
10 FORMAT (F6.3)
   DO 1 I = 1, 10
   READ (5, 10) (A(I,J), J = 1,10)
 1 CONTINUE
```

In either of these two examples, the subscript "I" is set to 1 and held there as "J" takes on values from 1 to 10. Then, "I" is set to 2 and held there as "J" again cycles from 1 to 10. The looping continues in this manner until both DO loops are satisfied (and 100 cards have been read). The values on the first 10 cards are assigned to A(1,1), A(1,2), A(1,3) . . . . A(1,10). The values on the second 10 cards are assigned to A(2,1), A(2,2) . . . . . . (A(2,10), and so on through the data cards. Note particularly that the ordering of elements in memory is <u>by row</u> instead of by column. This illustrates the fact that the programmer can manipulate the data into and out of arrays as needed by using subscripted arrays.

If 10 values were punched on each card, the following statements could be used (note the use of two IMPLICIT DO loops, carefully grouped by parentheses):

```
10 FORMAT (10F6.3)
   READ (5, 10) ((A(I,J),J=1, 10), I=1, 10)
```

Again, the subscript "I" increments once as "J" varies from 1 to 10.

The printing of an array is handled in the same manner. Suppose that a 40 x 8 array is stored in memory. It could be printed using:

```
20 FORMAT (8F14.3)
   WRITE (6,20) ((C(M,N), N=1,8), M=1,40)
      or
20 FORMAT (8F14.3)
   DO 2 M=1,40
   WRITE (6,20) (C(M,N), N=1,8)
 2 CONTINUE
```

Either of these combinations results in the following printout:

```
C(1,1)     C(1,2)     C(1,3)     C(1,4) . . . . . . . . . . C(1,8)
C(2,1)     C(2,2)     C(2,3)     C(2,4) . . . . . . . . . . C(2,8)
C(3,1)     C(3,2)     C(3,3)     C(3,4) . . . . . . . . . . C(3,8)
,
,
C(40,1)    C(40,2)    C(40,3)    C(40,4) . . . . . . . . . C(40,8)
```

CHAPTER 6

EXERCISES

1. Fill in the blanks:

   a. Variables that refer to an array of numbers are called __________ variables.

   b. A __________ is used to reference a specific __________ in an array. The 16th __________ of an array K is referred to as __________.

   c. Constants and variables used to form subscripts must be __________.

   d. The value of a subscript must never be __________ nor __________ than the number of elements specified in a __________ statement.

2. Which of the following are invalid subscripts and why?

   a. K (TOTAL)
   b. M (.5)
   c. X (-J)
   d. NAME (K + 3)
   e. LIMA (-2)
   f. X (KOUNT)
   g. X (I)
   h. X (KOUNT + I)

3. Given the array N:

$$\begin{matrix} 6 & 5 & 4 & 3 & 10 \\ 4 & 6 & 5 & 0 & 6 \\ 2 & 3 & 4 & 2 & 0 \end{matrix}$$

   a. What is the value of the following:

      (1) N(1,3) __________
      (2) N(3,5) __________
      (3) N(2,3) __________
      (4) N(3,4) __________

   b. Write a DIMENSION statement for array N.

      ____

   c. List the elements of array N that are printed after the following statement has been executed–

      WRITE (6,2) (N(3,J), J = 1, 5)

CHAPTER 6

EXERCISES–Continued

4. Write a DIMENSION statement for each of the following:

   a. To reserve a storage for a 6 X 7 X 8 array called LOCO and a 70-element array called ITEM.

   b. To reserve storage for a 1 X 10 array called SUMS.

5. Given: DIMENSION K (100)
   Write a program segment that will obtain the sum of all numbers greater than 50 in array K. Have the sum placed in NSUM. Have your program set all elements less than or equal to 50 to 0.

COMM. | STATEMENT NUMBER | CONT. | FORTRAN STATEMENT

CHAPTER 6

EXERCISES–Continued

6. Write a program segment that will examine each element of an array KOUNT (50) and add 15 to any element containing 21 or more; maintain a count in KTOT of all elements that are less than 21 and set those elements to 100.

COMM.	STATEMENT NUMBER	CONT.	FORTRAN STATEMENT
1	2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52

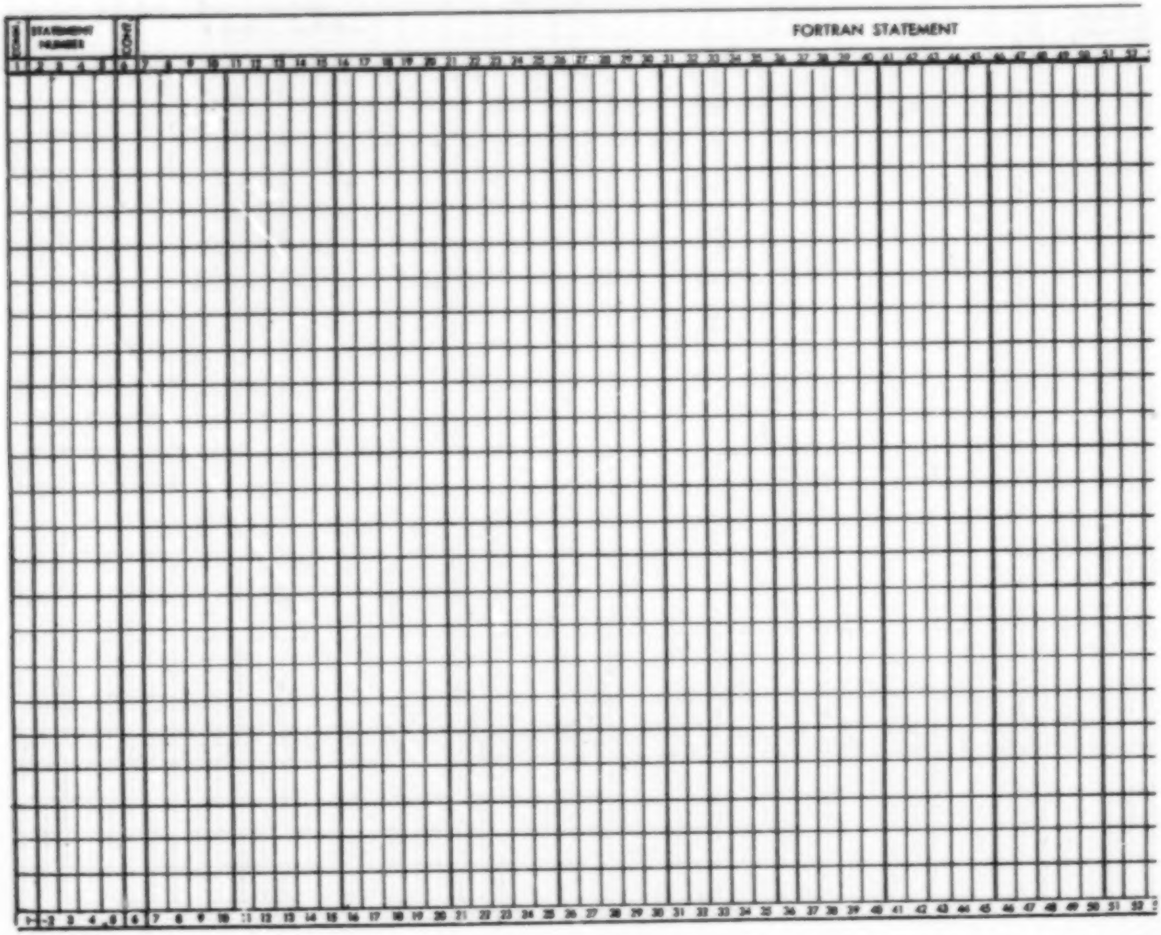
CHAPTER 6

EXERCISES–Continued

7. Given: The following data elements which are punched (one per card) in columns 1 through 4, exactly as shown.

.315
.280
.345
.310
.322

Write a complete program that will read this data into an array named AVG, sort the array from high to low, and then print the first three elements on successive print lines under the heading TOP THREE.



CHAPTER 6

EXERCISES–Continued

8. Given:
```
INTEGER FAILNO, NAMES, RATING
DIMENSION NAMES (50), RATING (50)
FAILNO = 0
```
The arrays have already been read into memory.

Write a program segment that will examine each rating in the RATING array. If the rating is greater than or equal to 70, print (using format number 2) the rating and corresponding name from the NAMES array. Otherwise, add 1 to FAILNO and then examine another element. After all ratings have been checked print FAILNO (using format number 3).

COMM. | STATEMENT NUMBER | CONT. | FORTRAN STATEMENT

CHAPTER 6

EXERCISES–Continued

9. There are 10 values on a card. The format statemnt is 2 FORMAT (10I4). Write the statements needed to store these values in array N(2,5) in the following order: (1,1), (1,2), (1,3), (1,4), (1,5), (2,1), (2,2) . . . . (2,5).

COMM. STATEMENT NUMBER CONT. FORTRAN STATEMENT

CHAPTER 6

EXERCISE SOLUTIONS

1. a. subscripted
   b. subscript, element, element, K(16)
   c. unsigned
   d. less than one, greater, DIMENSION

2. a. valid (if TOTAL is integer)
   b. invalid (subscript cannot be less than one)
   c. invalid (subscript must be unsigned)
   d. valid
   e. invalid (subscript must be unsigned)
   f. valid
   g. valid
   h. invalid (A variable cannot be added to another variable in a subscript.)

3. a. (1) 4
   (2) 0
   (3) 5
   (4) 2

   b. DIMENSION N(3,5)

   c. 2,3,4,2,0

4. a. DIMENSION LOCO (6,7,8),ITEM(70)
   b. DIMENSION SUMS(10)

5.
```
      DIMENSION K(100)
      NSUM = 0
      DO 20 I = 1, 100
      IF (K(I) - 50) 10, 10, 15
   10 K(I) = 0
      GO TO 20
   15 NSUM = NSUM + K(I)
   20 CONTINUE
```

CHAPTER 6

EXERCISE SOLUTIONS–Continued

6.

```
      DIMENSION KOUNT(50)
      KTOT = 0
      DO 20 J = 1, 50
      IF (KOUNT(J).GE.21) GO TO 10
      KTOT = KTOT + 1
      KOUNT(J) = 100
      GO TO 20
   10 KOUNT(J) = KOUNT(J) + 15
   20 CONTINUE
```

7.

```
      DIMENSION AVG(5)
    1 FORMAT(F4.3)
    2 FORMAT(1H1,9HTOP THREE//F4.3/F4.3/F4.3)
      DO 10 I = 1,5
   10 READ(5,1) AVG(I)
      DO 20 J = 1,4
      IP = J + 1
      DO 30 K = IP, 5
      IF (AVG(J).GT.AVG(K)) GO TO 30
      TAVG = AVG(J)
      AVG(J) = AVG(K)
      AVG(K) = TAVG
   30 CONTINUE
   20 CONTINUE
      WRITE (6,2) (AVG(I), I = 1,3)
      STOP
      END
```

8.

```
      INTEGER FAILNO, NAMES, RATING
      DIMENSION NAMES(50), RATING(50)
      FAILNO = 0
      DO 10 I = 1, 50
      IF (RATING(I) - 70) 5,6,6
    5 FAILNO = FAILNO + 1
      GO TO 10
    6 WRITE (6,2) NAMES(I), RATING(I)
   10 CONTINUE
      WRITE (6,3) FAILNO
      STOP
      END
```

9.

```
      READ (5,2) ((N(I,J), J=1,5), I=1,2)
              or
      DO 6 I = 1,2
      READ (5,2) (N(I,J), J=1,5)
    6 CONTINUE
```

CHAPTER 7

ALPHANUMERIC DATA

7.1 Introduction

Since most processing accomplished with the FORTRAN language is based upon calculations, FORTRAN, by design, does not lend itself well to the manipulation of alphanumeric data. Usually the only alphanumeric data in a FORTRAN program are headings and other constant, printed data. Nonetheless, from time to time it is necessary to input and output alphanumeric data and to evaluate alphanumeric data as part of the processing requirements.

Another factor that enters into consideration is the number of characters that may be stored in one computer "word" of storage. For purposes of illustration, the IBM standard of four characters per word is used.

7.2 Alphanumeric (A) Specification

This specification is used for input and output of alphabetic or alphanumeric data. The general format of this specification in a FORMAT statement is–

$$\boxed{\mathrm{n\,A\,w}}$$

where

n	is the number of computer words being described; if no value is given for n, the value is taken to be one.
A	indicates an alphanumeric field.
w	is the number of characters to be stored in one computer word.

The A FORMAT specification is used whenever data are to be transferred without conversion. Data which are read under A-type specification may be stored in variables (or variables which are arrays) of any type, but may not be used in computations or in comparisons which require conversion. As an example of the latter case, an integer variable containing characters read under A-type transmission cannot be compared with a real variable, as this would involve conversion and would probably result in a program ending abnormally. If two integer variables which contain data are read in under A-type specification, then they may be compared to each other, but usually only for equality.

Refer to figure 7-1 for an illustration of the use of the A-type specification. When the w in the specification is longer than the variable in the I/O list of a READ statement, only the rightmost characters are stored in the variable. If the variable is longer than the w in the specification, excess positions to the right are filled with blanks. If, during output, the variable in the I/O list of a WRITE statement is longer than the number of characters that are being transferred, the characters (w) are right-justified in the variable, preceded by leading blanks. If there are more characters (w) than positions in the variable, the excess characters to the right are truncated.

A common use of the A-type specification is to read character data into arrays. The primary purpose of this technique is to avoid naming consecutive words of storage, in order to accommodate lengthy strings of alphanumeric data. Remember the restriction of four characters per word used as an illustration. For example, consider the problem of reading a 20-character field. A FORMAT specification of A20 will not work as one variable, because the FORMAT description is restricted to the word size of the computer. It is possible to describe the 20 characters as 5A4, but five variable names are required in the I/O list of the READ statement.

Instead of multiple variable names, a better approach allows an array to define as many words as necessary to accommodate the 20-character field. Figure 7-2 provides an example of using an array to store alphanumeric data. The techniques of array control for alphanumeric data are the same as those discussed in Section 6.7, Reading and Printing an Array.

There are, of course, times when the alphanumeric field is not an even multiple of four-character groups. As long as the format specification, such as A2 or A3, is identical in both the input and output formats, it is not

Statement	Explanation
7 FORMAT(A3,A4) READ(8,7)FLDA,FLDB	As a result of the READ statement, data from the input device referenced by '8' will be placed in FLDA and FLDB. FLDA will receive 3 characters and FLDB will receive 4 characters.

Figure 7-1.—Alphabetic input.

Statement	Explanation
DIMENSION NAME(5) 42 FORMAT(5A4) READ(8,42)NAME . . WRITE(6,42)NAME	5 words of storage are defined by the DIMENSION statement. 4 characters of the record are stored in each word.
Example:	The following example shows an alphabetic name as input. It will be stored in 5 words of storage.
JONATHAN DONOTHING	JONA|THAN|ƀDON|OTHI|NGƀƀ

Figure 7-2.–Using an array for alphabetic input.

necessary to have a four-character field specification for each word represented in the array. (See figure 7-3.)

7.3 DATA Initilization Statement

A DATA statement is used to provide initial values for variables, arrays, and array elements. A DATA statement is non-executable and may appear in a program unit anywhere after the specification statements, if any specification statements are used. Figure 1-4 shows this relationship. All initially defined entities are defined for use when the program begins execution.

The format of DATA statement is;

```
DATA nlist /clist/ [[,]nlist/clist/]...
```

where

DATA	must appear as shown.
nlist	is a list containing the names of variables, arrays, or array elements to be initialized. The entries of the nlist are separated by commas.
/	must appear as shown.
clist	is a list of constants (integer, real, or literal). Any of these constants may be preceded by i*, where i is an unsigned integer constant. When the form i* appears before a constant, it indicates that the constant is to be specified i times. The constants of the clist are separated by commas.
/	must appear as shown.

Statement	Explanation
DIMENSION ALPHA(5) 6 FORMAT(5A4) READ(8,6)(ALPHA(I),I=1,5)	The 5 words of storage defined as the array ALPHA are initialized by the implied DO of the READ statement.
DIMENSION NAME(15) 6 FORMAT(15A2) READ(8,6)(NAME(K),K=1,15)	A total of 30 characters will be stored in NAME. It is not necessary for the FORMAT specification to be a multiple of 4.

Figure 7-3.–Using the IMPLICIT DO loop for alphabetic input.

There must be the same number of items specified by each nlist and its corresponding clist. There is a one-to-one correspondence between the items specified by nlist and the constants specified by clist, such that the first item of nlist corresponds to the first constant of clist.

The type of the nlist entity and the type of the corresponding clist constant must agree.

Examples of the DATA statement are provided in figure 7-4.

Example 1:

```
DIMENSION TAB(5)
DATA A,B,ICODE,TAB/4.7,0.0,‘Z’,5*1.5/,K/1/
```

The first nlist of A,B,ICODE,TAB is initialized with the corresponding values of A=4.7, B=0.0, ICODE = the character Z, and each of the five elements of the array TAB are initialized with the value 1.5. The second nlist contains the integer variable K that is set equal 1 at the beginning of execution of the program.

Example 2:

```
DIMENSION DAYS(7)
DATA DAYS/‘SUN ’,‘MON ’,‘TUES’,‘WED ’,‘THUR’,‘FRI ’,‘SAT ’/
```

This illustrates initializing an array with character data. Note the consideration for the four-character word length.

Figure 7-4.–DATA statement.

CHAPTER 7

EXERCISES

1. Code the FORMAT statement that will describe an input record that contains the following fields of information.

Positions	1-7	ID Number (numeric)
	8-10	Department Number (Alphabetic)
	11-28	Name (Alphabetic)
	29-100	Unused

2. Code the DIMENSION, FORMAT, READ, and WRITE statements required to utilize an array for the input and output of an 80 character string of alphabetic data.

3. Code a DATA statement to initialize the integer variable MILE to the value 5280.

4. Code one DATA statement to initialize the following:

    1. Each of three elements of the array AZ to the value zero.

    2. The real variable DIME to the value .10.

CHAPTER 7

EXERCISE SOLUTIONS

1. `2 FORMAT (I7,A3,6A3)`

2.
   ```
         DIMENSION CHAR(20)

       4 FORMAT(20A4)

         READ(5,4)(CHAR(I),I=1,20)

         WRITE(6,4)(CHAR(I),I=1,20)
   ```

3. `DATA MILE/5280/`

4. `DATA AZ(1), AZ(2), AZ(3), DIME/0.0,0.0,0.0,.10/`

   or

   `DATA AZ(1), AZ(2), AZ(3), DIME/3*0.0,.10/`

APPENDIX I

GLOSSARY

ALPHABETIC CHARACTER.–One of the letters A through Z.

ALPHAMERIC CHARACTER.–One of the characters A through Z and 0 through 9.

ALPHANUMERIC CHARACTER.–Any permissible character; letters, digits and others such as punctuation marks.

ARGUMENT.–An independent variable passed between a calling program and a subprogram.

ARITHMETIC EXPRESSION.–A grouping of arithmetic operands and operators.

ARRAY.–An arrangement of elements in one or more dimensions.

ARRAY ELEMENT.–One of the items of data within an array.

ASSEMBLE.–(1) To translate a program expressed in an assembly language into a computer language and perhaps to link subroutines. Assembling is usually accomplished by substituting the computer language operation code for the assembly language operation code and by substituting absolute addresses, immediate addresses, relocatable addresses, or virtual addresses for symbolic addresses. (2) To prepare a machine language program from a symbolic language program by substituting absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses. (Contrast with compile.)

ASSEMBLER.–A computer program used to assemble. Synonymous with assembly program.

ASSIGNMENT STATEMENT.–An instruction consisting of a variable, followed by an equal sign (=), followed by an arithmetic expression.

BASIC REAL CONSTANT.–A string of decimal digits containing a decimal point.

CALL STATEMENT.–Used to invoke or begin execution of a SUBROUTINE subprogram.

CALLING PROGRAM.–A program that transfers control to a specific closed subprogram.

COMPILE.–(1) To translate a computer program expressed in a problem-oriented language into a computer-oriented language. (2) To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler. (Contrast with assemble.)

COMPILE-AND-RUN.–An operating technique in which there are no stops between the compiling, loading, and execution of a computer program.

COMPILER.–Computer program used to compile. Synonymous with compiling program.

CONSTANT.–Data with a fixed value or meaning that is available for use throughout the program.

CONTROL STATEMENT.–One of the FORTRAN statements used to change the sequence of execution of a program.

DATA INITIALIZATION STATEMENT.–Used to define initial values of variables, array elements, and arrays. It is not executable.

DATA ITEM.–One element of information.

DATA SET.–A group of related data records.

DATA SET REFERENCE NUMBER.–A constant in a FORTRAN READ or WRITE statement that specifies the hardware device on which the data set resides.

DATA TYPE.–The internal form and mathematical characteristics of data. The two most common types are integer and real.

DELIMITER.–A flag that separates and organizes items of data.

DO LOOP.–A group of statements executed under the control of the DO statement.

DO VARIABLE.–A variable associated with a DO loop that is used to control the number of executions of the loop.

DUMMY ARGUMENT.–A variable associated with the subprogram statements FUNCTION or SUBROUTINE; they are matched with actual arguments from the calling program during execution.

END STATEMENT.–The statement that identifies the physical end of a FORTRAN source program.

ENTITY.–Something that exists independently, not relative to other things; a discrete unit.

EXECUTABLE PROGRAM.–An independent program.

EXECUTABLE STATEMENT.–A FORTRAN statement that causes some action to be taken.

EXPONENTIATION.–The raising of a quantity to a power.

EXTERNAL FUNCTION.–A function that is defined outside of the program that refers to it.

EXTERNAL PROCEDURE.–A subprogram or procedure defined by another language instead of FORTRAN.

FILE.–A collection of related data records organized in a specific manner. For example, an inventory file (one record for each inventory item).

FORMAT STATEMENT.–Statement is used in conjunction with certain input/output statements to specify the form in which data appears in a FORTRAN record on an input/output device.

FORMATTED RECORD.–A record which was written using a FORMAT statement.

FUNCTION SUBPROGRAM.–A FORTRAN subprogram that begins with a FUNCTION statement.

HIERARCHY OF OPERATIONS.–Relative precedence of execution of arithmetic operators.

HIGH-LEVEL LANGUAGE.–A programming language that does not reflect the structure of any one given computer or that of any given class of computers.

HOLLERITH.–Pertaining to a particular type of code or punched card utilizing 12 rows per column and usually 80 columns per card.

IMPLIED DO.–The use of implied looping control similar to a DO loop, without the word DO, operating on a list of variables rather than a series of statements.

INPUT/OUTPUT STATEMENTS.–Statements which, in addition to controlling input/output devices, enable the user to transfer data between internal storage and an input/ output medium.

INTEGER CONSTANT.–A series of decimal digits containing no decimal point.

I/O LIST.–The list of variable names associated with a READ or WRITE statement, that specify storage locations into which data is to be read from or from which data is to be written.

LITERAL CONSTANT.–An unchanging series of alphanumeric characters either enclosed in single apostrophes or preceded by the wH (Hollerith) specification.

LOGICAL OPERATOR.–One of the logical conjunctions .NOT., .AND., .OR..

LOOP.–A set of instructions that may be executed repeatedly while a condition prevails. Usually under control of a DO statement.

MAIN PROGRAM.–A program that does not contain the subprogram statements FUNCTION or SUBROUTINE but does contain at least one executable statement.

NAME.–A group of characters used to identify a program, subprogram, variable or an array. In FORTRAN it consists of from one to six alphameric characters–the first character must be alphabetic.

NESTED DO.–A DO loop contained within the range of another DO loop.

NONEXECUTABLE STATEMENT.–A statement which provides information for the operation of the program. The FORMAT statement provides data characteristics and editing information.

NUMERIC CHARACTER.–One of the characters 0 through 9.

NUMERIC CONSTANT.–An integer or real constant.

OBJECT PROGRAM.–A fully compiled or assembled program that is ready to be loaded into the computer. (Contrast with source program.)

PREDEFINED SPECIFICATION.–A FORTRAN standard defining the type of a variable depending on the first letter of the variable name. INTEGER variables begin with the letters I through N, REAL variables begin with A through H and O through Z.

PROCEDURE SUBPROGRAM.–A FUNCTION or SUBROUTINE subprogram.

PROGRAM.–A series of statements of a given language, written in a specific order; designed to accomplish a particular objective.

PROGRAM UNIT.–A main program or subprogram.

RANGE OF A DO STATEMENT.–Those statements that follow a DO statement, including the statement specified in the DO statement as the last of the loop.

READ.–To acquire or interpret data from a storage device, from a data medium, or from another source.

READ STATEMENT.–The FORTRAN statement that transfers data from an external source into internal memory.

REAL CONSTANT.–A string of decimal digits containing a decimal point.

RECORD.–A collection of related items of data, treated as a unit; a complete set of such records may form a file.

RELATIONAL EXPRESSION.–An expression that compares two arithmetic expressions providing a true or false condition.

RELATIONAL OPERATOR.–One of the six possible comparisons that can exist between two arithmetic expressions. The operators are .GT., .GE., .LT., .LE., .EQ., and .NE., and are greater than, greater than or equal to, less than, less than or equal to, equal to, and not equal to, respectively. The comparisons are evaluated as true or false.

RETURN STATEMENT.–Used as the logical end of a FORTRAN source program.

SOURCE PROGRAM.–A series of statements written in a programming language such as FORTRAN or COBOL; input to a compiler or assembler. (Contrast with object program.)

SOURCE LISTING.–A printed copy of all of the source language statements of a source program that were processed by a compiler or assembler.

SPECIFICATION STATEMENT.–A statement that provides the FORTRAN compiler with information about the data used in the source program.

STATEMENT.–A meaningful expression that may describe or specify operations. It consists of names, operators, constants and predefined words.

STATEMENT NUMBER.–A one to five digit number placed in columns 1 through 5 of the first line of a statement. It provides a unique identity to the statement. The number must be non-negative and non-zero.

STOP STATEMENT.–Stops execution of the program whenever executed. It serves as a logical terminator.

SUBPROGRAM.–A program that begins with the FUNCTION or SUBROUTINE statement.

SUBPROGRAM STATEMENT.–Statements which enable the user to name and to specify arguments for functions and subroutines.

SUBROUTINE.–A sequenced set of statements that may be used in one or more computer programs and at one or more points in a computer program.

SUBROUTINE SUBPROGRAM.–A subprogram that begins with the SUBROUTINE statement.

SUBSCRIPT.–An integer constant or integer variable or certain combinations of both, enclosed in parentheses, associated with an array name, used to identify a particular array element.

TRUNCATION.–The loss of data from a leading or trailing position in accordance with specified criteria.

TYPE DECLARATION.–An explicit statement of the type of a variable, array, or function by the use of an explicit specification statement.

UNFORMATTED RECORD.–A record created without the use of a FORMAT statement.

VARIABLE.–A character or group of characters that refers to a value and, in the execution of a computer program, corresponds to an address.

WRITE.–To record data in a storage device or a data medium.

WRITE STATEMENT.–The FORTRAN statement that transfers data from internal memory to a storage device or a data medium.

APPENDIX II

COMPLETE FORTRAN PROBLEM

This appendix includes a typical problem statement a programmer could receive as an assignment. A flowchart has been developed to solve the problem and the program written from the flowchart has been cross-referenced to show how the program statements correspond to the symbols of the flowchart.

No particular attempt has been made to write the "most efficient" or "best" or "fastest" or "smallest" FORTRAN program. Instead, a representative approach has been applied to allow for the widest possible understanding of each technique presented. It is of primary importance when preparing a program to be correct first and then optimize rather than waste time and effort attempting to be clever or exotic.

Problem Statement

The weather station at NAS ALERT routinely collects each day's highest temperature and other pertinent facts concerning the weather conditions. The need for a new ADP report has been identified. This report should provide the highest and lowest temperature in degrees Fahrenheit of the year, the average temperature, and the average temperature in degrees Celsius for any calendar year. An appropriate heading that includes the base name and the year being processed should be provided for this single page report.

Analysis

It can be determined that the program should be flexible enough to process the temperatures from any calendar year, and/or recording station so a method of introducing the year and the recording station name is required.

There are enough facts recorded for each day's weather to make it feasible to prepare an individual record for each day's weather conditions. Although this problem requires only the temperature from each day, it may be wise to investigate further uses of the other information that is available; this could very well save other data collection efforts later.

The analyst/programmer has decided to prepare as the first data record a record containing the year to be processed and the recording station name, followed by individual records containing each day's temperature and any other data deemed relevant. Leap years provide a slight complication to this procedure. It is necessary to determine if the year being processed is a leap year so that action may be taken to allow the program to handle the 366th day's information.

The Fahrenheit to Celsius conversion is a procedure that is frequently required and would be ideally suited as a subprogram. The subprogram would need to receive a temperature in Fahrenheit, perform the conversion, and pass the Celsius temperature back to the main or calling program.

The format of the printed report has not been specified and is left up to the discretion of the programmer.

Narrative

This narrative coordinates the efforts of the flowchart and the FORTRAN program that was developed from the flowchart. Each flowchart symbol is identified by the letter enclosed in parentheses to the left of the symbol. The program statement label numbers are used for identification of the various program segments. The flowchart reads from top to bottom, left to right, simply to fit better on a single page. Also keep in mind that this flowchart probably does not solve the problem in exactly the manner you would have used. The more important fact to consider is that the flowchart correctly solves the problem. In some cases, an alternative to the approach that was taken is given for comparison.

Record Formats:

The record which contains the recording station name and year being processed has the following format:

Positions 1-4	Year (1976, 1979, etc.)
Positions 5-24	Recording Station Name
Positions 25-80	Unused

The temperature record format is as follows:

Positions 1-3	Fahrenheit temperature (82, -15, 102)
Positions 4-7	Precipitation amount (two decimal places)
Positions 8-9	Unused
Positions 10-20	Cloud condition (partly cloudy, sunny, etc.)
Positions 21-80	Unused

COMPARISON NARRATIVE

Flowchart Symbol	Statement Numbers	Explanation
A		The standard beginning symbol for any flowchart.
B	Comment lines	Although FORTRAN does not have an Identification Division similar to a COBOL program, it is important to identify in some manner, the purpose, the author, the date written, and the other relevant data to the program. This is best accomplished through the use of comment lines at the beginning of the program. Comment lines may also be inserted to separate functional segments of coded statements or to explain particularly difficult areas.
B		The INTEGER statement defines the variables YEAR and DAYCNT and the array TEMPS as INTEGER type. These names offer a descriptive meaning. Each of the names could have been preceded by one of the letters J, K, L, M, or N to make the variables and array elements INTEGER, or some other name could have been used. The compiler used for this run does not allow statement numbers for the INTEGER and DIMENSION statements.
B		The DIMENSION statement defines the array TEMPS, each element of which will hold one day's temperature, and the array BASE, which will hold the alphabetic name of the base being processed. The arrays could have appeared in reverse order.
B	40-76	These FORMAT statements define the input and output for the program. Line 40 defines the year and base record. Line 50 defines the input temperature records. All this program really needs to have described is the first I3 field, which is the temperature. The other field specifications are not needed and could have been left off. Statements 60 through 76 are for the output report. Note the use of both Hollerith (H) and literal (enclosed in apostrophes) methods of presenting alphanumeric constant data. These FORMAT statements could have been placed next to their respective READ or WRITE statements. It is the programmer's choice.

Flowchart Symbol	Statement Numbers	Explanation
B	78	Statement 78 sets an initial value to the variable DAYCNT. This variable is used later for control of loops.
C	80	Statement 80 reads the first input record, which contains the year to be processed and the recording station at which the temperatures were recorded.
D	90-110	Statement 90 determines if the year is a leap year, in which case one more day's temperature must be included in all processing. Leap year is any year evenly divisible by 4. By using INTEGER arithmetic, any remainder from the division of the year by 4 will be lost, and when the quotient is multiplied by 4 the product will be less than the year at the beginning of the expression. For example, use the year 1976. YEAR = 1976 YEAR/4 = 494 494 * 4 = 1976 YEAR - 1976 = 0 The IF statement causes control to pass to statement 100 if the expression is evaluated as zero which, in this case, it is. Statement 100 adds 1 to DAYCNT to allow for the extra day in a leap year. When the expression is evaluated as other than zero, the IF statement passes control to statement 110 and DAYCNT stays at its initial value of 365. Try 1979. YEAR = 1979 1979/4 = 494, remember, no remainders in INTEGER arithmetic 494 * 4 = 1976 YEAR - 1976 = 3

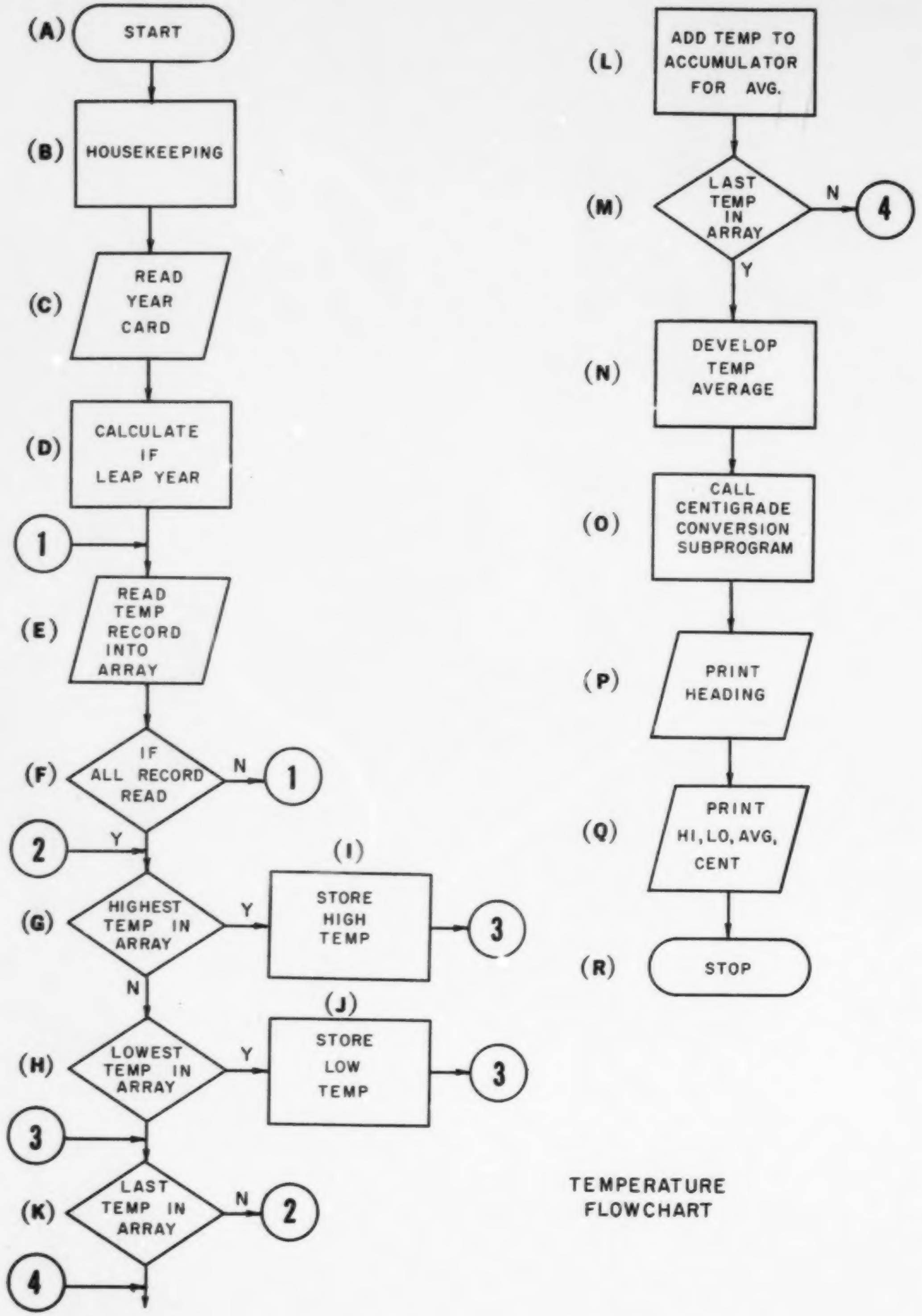
Flowchart Symbol	Statement Numbers	Explanation
E & F	120-130	This DO loop causes the READ statement to be executed DAYCNT number of times. DAYCNT is either 365 or 366, depending on the results of the leap year calculation performed earlier. The value of I begins at 1; the temperature in the first weather record is placed in the first element of the array TEMPS. When the DO loop is executed the second time, I equals 2 and the second weather record's temperature is placed in the second element of TEMPS. Only the array name TEMPS is associated with the READ statement, therefore only the first field specification of the FORMAT statement on line 50, I3, is used to describe the element of the array TEMPS. The CONTINUE statement on statement 130 could have been left out, as could the incremental value for the DO loop, since the increment is 1 if not specified; the DO loop then could be coded DO 125 I = 1, DAYCNT.
F		Symbol F of the flowchart corresponds to the DO loop internal procedure of checking the contents of I against the value of DAYCNT in order to determine completion of the proper number of executions of the loop.
G, H, I, J, K	140-240	These segments of the flowchart and program provide the search for the highest and lowest temperatures stored in the array TEMPS.
	140	Statement 140 sets the variable I to 1. I will continue to be used as a subscript for the array TEMPS.
	150-160	In order to find the highest and lowest temperature in the array, it is logically necessary to compare each of the temperatures in the array to every other temperature. The program begins this comparison at the beginning or first element of the array. As a logical beginning, the first temperature in the array is stored in areas designated for both the highest and lowest temperatures. At this point in the program, statement 160, we know that the first element contains both the highest and lowest temperatures ENCOUNTERED SO FAR!
	170	Statement 170 adds 1 to the value of I. The first time this is executed, I is equal to 2.

Flowchart Symbol	Statement Numbers	Explanation
	180	This statement compares the temperature in the second element of the array against the value in LHIGH, which contains the highest value ENCOUNTERED SO FAR! If the temperature in the array is GREATER than the temperature in LHIGH, control passes to statement 210.
I	210	Statement 210 replaces the value in LHIGH with the higher temperature in the second element of the array.
	220	Statement 220 then passes control to statement 240.
	240	Statement 240 determines if all the temperatures in the array have been compared. In this, the first pass through the program, I = 2 causes control to return to statement 170, where the comparison process is executed again.
G	180	Return to statement 180 and symbol G. If the temperature in the second element of the array was NOT greater than the temperature stored in LHIGH, then it may be LESS than the temperature stored in LOW.
H	190	Statement 190 compares the temperature in the second element of the array against the temperature stored in LOW. If the temperature in the array is lower, control passes to statement 230.
J	230	Statement 230 replaces the value in LOW with the value in the second element of the array. Control is then passed to statement 240.
K	240	Statement 240 determines if all the temperatures in the array have been compared. As mentioned before, in this first example (I = 2), control returns to statement 170 where the comparison process is repeated. When I does equal the value in DAYCNT, either 365 or 366, control passes to statement 245.

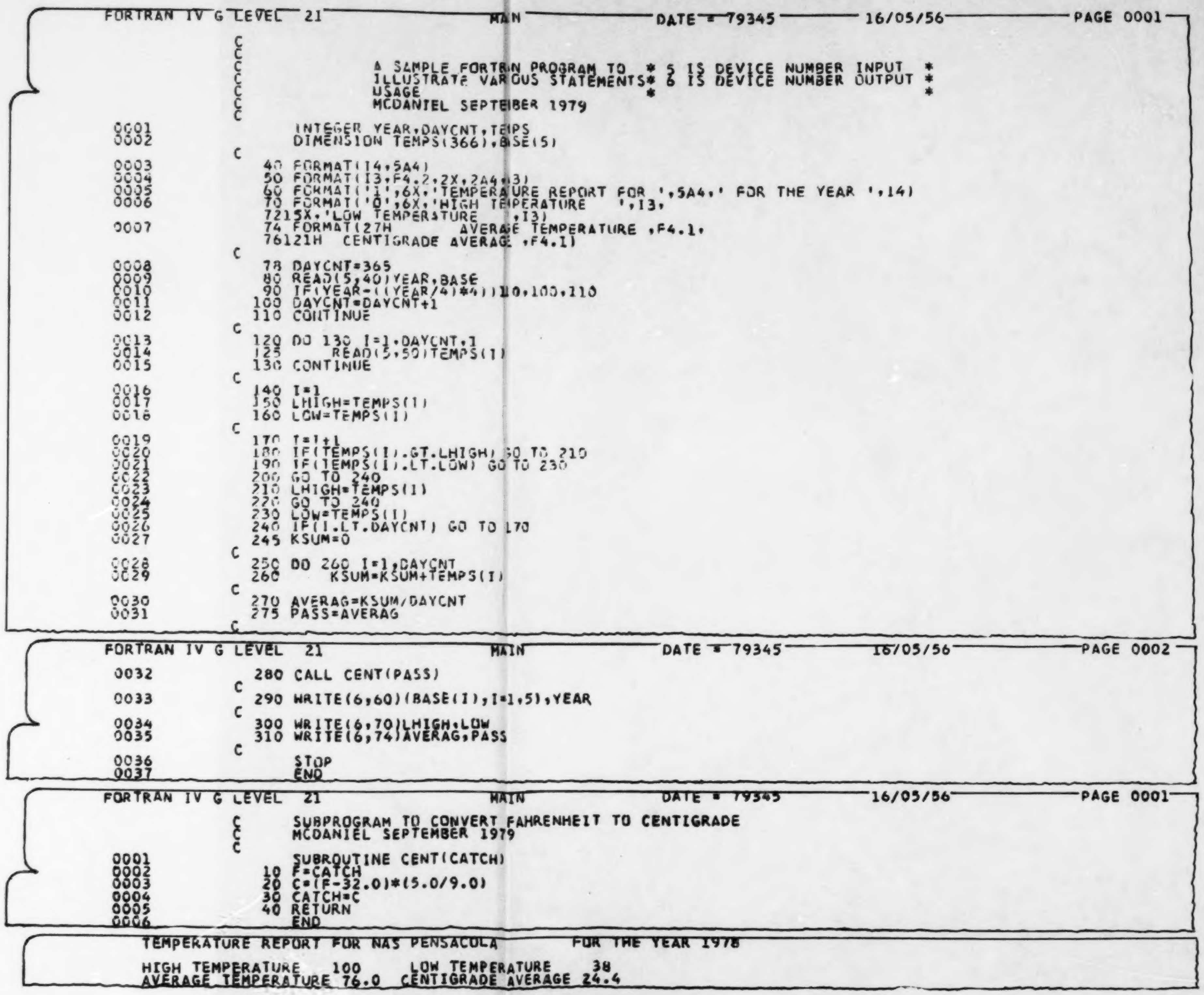
Flowchart Symbol	Statement Numbers	Explanation
	200	If neither statement 180 nor 190 is TRUE, the temperature in the array is not higher than LHIGH or lower than LOW, control is passed to statement 240, and that statement functions as previously mentioned.
		NOTE: The preceding narrative of statements 170 through 240 refers to the comparison of the second element against the values in LHIGH and LOW. As the loop is repeated, the value of I is incremented and this value, used as the subscript, determines which element is actually being referenced in statements 180, 190, 210, and 230. This loop, 170 through 240, could have been accomplished through the use of a DO loop.
L-M	245-260	Statement 245 initializes the variable KSUM to zero. KSUM is used to accumulate all the temperatures in preparation to develop the average temperature. This statement could have been located in the housekeeping area of the program.
	250-260	This DO loop executes the number of times specified by DAYCNT; adding the temperature in each element of the array to the variable KSUM.
N	270	The yearly average is developed.
O	275-280	The variable AVERAG now contains the computed yearly average. The variable to be passed to the subprogram is also used to return to the main program the Celsius temperature. Therefore, a variable other than AVERAG must be used in the CALL statement that invokes the subprogram. The variable PASS is set to the average temperature in AVERAG.
	280	This CALL statement causes the SUBROUTINE subprogram CENT to be executed at this point. The value contained in the variable PASS is made available to the subprogram in the corresponding variable associated with the SUBROUTINE statement of the subprogram. In this case, there is only one variable in each place; the value of the average temperature developed

Flowchart Symbol	Statement Numbers	Explanation
	280 (Continued)	in the main program can now be utilized by the subprogram simply by using the variable CATCH. At this time, the subprogram returns a Celsius temperature to the main program through the variable CATCH. More discussion on the subprogram follows. After execution of the subprogram is completed, control returns to the statement immediately following the CALL statement.
P	290	Statement 290 prints the heading at the top of the page. Note the '1' as the first character of FORMAT statement 60. The 'implied' DO loop is used to transfer the alphabetic recording station name from the array BASE to the output printer area. This is in contrast to the input technique used with the READ statement in statement number 80.
Q	300-310	Statement 300 causes the high and low temperatures to be printed two lines below the heading line. Statement 310 causes the average Fahrenheit temperature and the average Celsius temperature (in the variable PASS) to be printed one line below the line printed by statement 300.
R		The STOP statement signifies the logical end of the program. The END statement is the physical end of the program. The subprogram in this instance was small enough to preclude the need of a flowchart.
	10	At the beginning of execution of the subprogram CENT, the Fahrenheit average temperature is stored in CATCH. Statement 10 sets F equal to CATCH.
	20	The Celsius conversion is performed. Note the use of real variables and constants. This prevents any fractional loss during the division. The conversion formula for Fahrenheit to Celsius is 5/9 of the remainder of the Fahrenheit temperature minus 32°. The expression (5.0/9.0) has been used so as to be more easily related to the given formula. This expression could be reduced to the decimal value .555 in the calculation, but the documentation value of

Flowchart Symbol	Statement Numbers	Explanation
	20 (Continued)	(5.0/9.0) may have greater value for those who look at the program later than would a shorter arithmetic expression.
	30	This statement places the Celsius temperature in CATCH so that it is available to the main program when control is returned to the main program.
	40	Statement 40 causes control to be returned to the main program. The END statement marks the physical end of the subprogram. In this particular example, a FUNCTION subprogram may have been an alternative to consider. The differences are as follows: Statement 275 and 280 could have been replaced with one statement, CTEMP = CENT(AVERAG). The subroutine statement could have been replaced with FUNCTION CENT(CATCH). The rest of the FUNCTION subprogram could have been CENT=(CATCH-32.0)*.555 RETURN END



TEMPERATURE FLOWCHART



```
FORTRAN IV G LEVEL  21                  MAIN              DATE = 79345          16/05/56             PAGE 0001
                  C
                  C
                  C                A SAMPLE FORTRAN PROGRAM TO  * 5 IS DEVICE NUMBER INPUT  *
                  C                ILLUSTRATE VARIOUS STATEMENTS* 6 IS DEVICE NUMBER OUTPUT *
                  C                USAGE                        *                           *
                  C                MCDANIEL SEPTEMBER 1979
                  C
0001                    INTEGER YEAR,DAYCNT,TEMPS
0002                    DIMENSION TEMPS(366),BASE(5)
                  C
0003                 40 FORMAT(I4,5A4)
0004                 50 FORMAT(I3,F4.2,2X,2A4,I3)
0005                 60 FORMAT('1',6X,'TEMPERATURE REPORT FOR ',5A4,' FOR THE YEAR ',I4)
0006                 70 FORMAT('0',6X,'HIGH TEMPERATURE    ',I3,
                     7215X,'LOW TEMPERATURE    ',I3)
0007                 74 FORMAT(27H          AVERAGE TEMPERATURE ,F4.1,
                     76121H  CENTIGRADE AVERAGE ,F4.1)
                  C
0008                 78 DAYCNT=365
0009                 80 READ(5,40)YEAR,BASE
0010                 90 IF(YEAR-((YEAR/4)*4))110,100,110
0011                100 DAYCNT=DAYCNT+1
0012                110 CONTINUE
                  C
0013                120 DO 130 I=1,DAYCNT,1
0014                125     READ(5,50)TEMPS(I)
0015                130 CONTINUE
                  C
0016                140 I=1
0017                150 LHIGH=TEMPS(I)
0018                160 LOW=TEMPS(I)
                  C
0019                170 I=I+1
0020                180 IF(TEMPS(I).GT.LHIGH) GO TO 210
0021                190 IF(TEMPS(I).LT.LOW) GO TO 230
0022                200 GO TO 240
0023                210 LHIGH=TEMPS(I)
0024                220 GO TO 240
0025                230 LOW=TEMPS(I)
0026                240 IF(I.LT.DAYCNT) GO TO 170
0027                245 KSUM=0
                  C
0028                250 DO 260 I=1,DAYCNT
0029                260     KSUM=KSUM+TEMPS(I)
                  C
0030                270 AVERAG=KSUM/DAYCNT
0031                275 PASS=AVERAG
                  C
FORTRAN IV G LEVEL  21                  MAIN              DATE = 79345          16/05/56             PAGE 0002
 0032               280 CALL CENT(PASS)
                  C
 0033               290 WRITE(6,60)(BASE(I),I=1,5),YEAR
                  C
 0034               300 WRITE(6,70)LHIGH,LOW
 0035               310 WRITE(6,74)AVERAG,PASS
                  C
 0036                   STOP
 0037                   END
FORTRAN IV G LEVEL  21                  MAIN              DATE = 79345          16/05/56             PAGE 0001
                  C     SUBPROGRAM TO CONVERT FAHRENHEIT TO CENTIGRADE
                  C     MCDANIEL SEPTEMBER 1979
                  C
 0001                   SUBROUTINE CENT(CATCH)
 0002                10 F=CATCH
 0003                20 C=(F-32.0)*(5.0/9.0)
 0004                30 CATCH=C
 0005                40 RETURN
 0006                   END

       TEMPERATURE REPORT FOR NAS PENSACOLA        FOR THE YEAR 1978

       HIGH TEMPERATURE   100      LOW TEMPERATURE    38
       AVERAGE TEMPERATURE 76.0  CENTIGRADE AVERAGE 24.4
```

best copy available

INDEX

A

Alphanumeric data, 7-1 to 7-6
  alphanumeric (A) specification, 7-1 to 7-3
  DATA initialization statement, 7-3 to 7-4
  exercise solutions, 7-6
  exercises, 7-5
  introduction, 7-1

Analysis, FORTRAN problem, AII-1

Arithmetic, 3-1 to 3-19
  arithmetic expressions, 3-2 to 3-5
  arithmetic operations, 3-2
  arithmetic statements, 3-6
  exercise solutions, 3-18 to 3-19
  exercises, 3-15 to 3-17
  functions and subroutines, 3-7 to 3-14
    FORTRAN-supplied subprograms, 3-7
    FUNCTION subprograms, 3-9 to 3-11
    RETURN and END statements in a FUNCTION subprogram, 3-11
    SUBROUTINE subprograms, 3-12 to 3-14
    user-written subprograms, 3-9
  integer arithmetic, 3-1
  real (floating-point) arithmetic, 3-1

Arithmetic expressions, 3-2 to 3-5

Arithmetic IF statement, 5-1 to 5-3

Arithmetic operations, 3-2

Arithmetic statements, 3-6

Array, 6-6 to 6-11
  reading and printing, 6-7 to 6-11
  storing, 6-6

C

Carriage control input/output operations, 4-13
Character set, FORTRAN, 1-9
Coding form, FORTRAN, 1-5
Comparison narrative, FORTRAN problem, AII-3 to AII-9
Compile and run, FORTRAN, 1-5
Compiler, FORTRAN, 1-3
Complete FORTRAN problem, AII-1 to AII-9
  analysis, AII-1
  comparison narrative, AII-3 to AII-9
  narrative, AII-2
  problem statement, AII-1
  record formats, AII-2
Computed GO TO statements, 5-5 to 5-6
Constants, 2-2
Constants and variables, 2-1 to 2-8
  constants, 2-2
  exercise solutions, 2-8
  exercises, 2-7
  FORTRAN numbers, 2-1
  INTEGER and REAL statements, 2-5 to 2-6
  variables, 2-3 to 2-4
CONTINUE statement, 5-10
Control statements, 5-1 to 5-20
  arithmetic IF statement, 5-1 to 5-3
  computed GO TO statements, 5-5 to 5-6
  CONTINUE statement, 5-10
  DO statement, 5-7 to 5-10
  END statement, 5-12
  exercise solutions, 5-18 to 5-20
  exercises, 5-13 to 5-17
  GO TO statement (unconditional), 5-1
  introduction, 5-1
  relational IF statement, 5-3 to 5-5
  RETURN statement, 5-12
  STOP statement, 5-11

D

DATA initialization statement, 7-3 to 7-4
DIMENSION statement, 6-5
DO statement, 5-7 to 5-10
Double subscripts, 6-3

E

End-of-file concepts, 4-4
END statement, 5-12
Exercise solutions, 1-14, 2-8, 3-18 to 3-19, 4-19 to 4-21, 5-18 to 5-20, 6-18 to 6-19, 7-6
    chapter 1, FORTRAN, 1-14
    chapter 2, constants and variables, 2-8
    chapter 3, arithmetic, 3-18 to 3-19
    chapter 4, input/output operations, 4-19 to 4-21
    chapter 5, control statements, 5-18 to 5-20
    chapter 6, subscripted variables and arrays, 6-18 to 6-19
    chapter 7, alphanumeric data, 7-6
Exercises, 1-12 to 1-13, 2-7, 3-15 to 3-17, 4-15 to 4-18, 5-13 to 5-17, 6-12 to 6-17, 7-5
    chapter 1, FORTRAN, 1-12 to 1-13
    chapter 2, constants and variables, 2-7
    chapter 3, arithmetic, 3-15 to 3-17
    chapter 4, input/output operations, 4-15 to 4-18
    chapter 5, control statements, 5-13 to 5-17
    chapter 6, subscripted variables and arrays, 6-12 to 6-17
    chapter 7, alphanumeric data, 7-5

F

Format notation, 1-8
FORMAT statement, input/output operations, 4-6 to 4-12
FORTRAN, 1-1 to 1-14
    exercise solutions, 1-14
    exercises, 1-12 to 1-13
    FORTRAN programming, 1-1

FORTRAN–Continued
    information about the course, 1-3 to 1-11
        compile and run, 1-5
        format notation, 1-8
        FORTRAN character set, 1-9
        FORTRAN coding form, 1-5
        FORTRAN compiler, 1-3
        FORTRAN statements, 1-10 to 1-11
    introduction, 1-1
FORTRAN numbers, 2-1
FORTRAN problem, complete, AII-1 to AII-9
FORTRAN-supplied subprograms, 3-7
FUNCTION subprograms, 3-9 to 3-11
Functions and subroutines, arithmetic, 3-7 to 3-14
    FORTRAN-supplied subprograms, 3-7
    FUNCTION subprograms, 3-9 to 3-11
    RETURN and END statements in a FUNCTION subprogram, 3-11
    SUBROUTINE subprograms, 3-12 to 3-14
    user-written subprograms, 3-9

G

Glossary, AI-1 to AI-5
GO TO statements, computed, 5-5 to 5-6
GO TO statement (unconditional), 5-1

I

IF statement, 5-1 to 5-5
    arithmetic, 5-1 to 5-3
    relational, 5-3 to 5-5
Input/Output operations, 4-1 to 4-21
    carriage control, 4-13
    exercise solutions, 4-19 to 4-21
    exercises, 4-15 to 4-18
    FORMAT statement, 4-6 to 4-12
    introduction, 4-1
    multiple record format statement, 4-13 to 4-14
    READ statement, 4-2 to 4-3
        end-of-file concepts, 4-4
    WRITE statement, 4-5
INTEGER and REAL statements, 2-5 to 2-6
Integer arithmetic, 3-1

M

Multiple record format statement, 4-13 to 4-14

N

Narrative, FORTRAN problem, AII-2

R

READ statement, input/output operations, 4-2 to 4-3
  end-of-concepts, 4-4
Reading and printing an array, 6-7 to 6-11
Real (floating-point) arithmetic, 3-1
Record formats, FORTRAN problem, AII-2
Related data, grouping of, subscripted variables and arrays, 6-1
Relational IF statement, 5-3 to 5-5
RETURN and END statements in a FUNCTION subprogram, 3-11
RETURN statement, 5-12

S

Specification, alphanumeric (A), 7-1 to 7-3
Statements, FORTRAN, 1-10 to 1-11
STOP statement, 5-11
Storing of an array, 6-6
SUBROUTINE subprograms, 3-12 to 3-14
Subscripted variables and arrays, 6-1 to 6-19
  DIMENSION statement, 6-5
  double subscripts, 6-3
  exercise solutions, 6-18 to 6-19
  exercises, 6-12 to 6-17
  grouping of related data, 6-1
  reading and printing an array, 6-7 to 6-11
  storing of an array, 6-6
  subscripts, 6-2
  three-dimensional arrays, 6-4
Subscripts, 6-2

T

Three-dimensional arrays, 6-4

U

User-written subprograms, 3-9

V

Variables, 2-3 to 2-4

W

WRITE statement, input/output operations, 4-5